Software Verification for Sensor Nodes

Doina Bucur

September 2, 2010

Abstract

We look at software written for wireless sensor nodes, and specialize and develop on the state of the art in software verification techniques for standard C programs (CBMC and SatAbs from the CProver suite) in order to locate programming errors in sensor applications before the software's deployment on motes. Ensuring the reliability of sensor applications is an exemplary difficult problem for the validation and verification field: low-level, interrupt-driven or multithreaded code runs without memory protection in dynamic environments, and the difficulties lie with:

1. being able to automatically extract standard C models out of the particular flavours of embedded C used for one sensor programming solution or another,
2. decreasing the program's state space to a degree that allows the resulting tool to be practically useful, and
3. writing specifications which capture non-trivial behaviour.

This report describes two such approaches.

One is a software verification tool for platform-independent, OS-dependent sensor programs: multithreaded TinyOS applications written over the C TOSThreads library. This approach of verifying one module at a time naturally decreases the program's state space, such that—even as the application is multithreaded—safety specifications over fairly complex monitoring applications can be verified in under one hour's time. In order to automatically extract a correct model out of the single TOSThreads module, we need to manually construct brief models of those OS driver modules called by the application. The resulting program is then passed to SatAbs [8], a software verifier for multithreaded ANSI C, with specifications (written as C assertions) pinpointing incorrect interface use and incorrect adaptation to environmental input.

The other is a platform-dependent, OS-independent orthogonal approach: a software verification tool for large, OS-wide programs written in MSP430 embedded C with asynchronous hardware interrupts. Our tool automatically translates the program into standard C by replacing direct memory access with a model of the MCU's memory map. A number of calls to hardware interrupt handlers are inserted into the main application to emulate the existence of hardware, and their occurrence is minimized with a partial-order reduction technique, in order to decrease the program's state space. Safety specifications are written as C assertions embedded in the code. The resulting sequential program is then passed to CBMC [7], a bounded software verifier for sequential ANSI C. Besides memory-related errors (e.g., out-of-bounds arrays, null-pointer dereferences), this tool chain verifies 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. To the publication date of our tools [4, 5], little specialized work existed on the topic; we overview both background and subsequent work on the topic and draw a comparison.

Contents

1 Introduction and Background **4**
1.1 Introduction . . . 4
1.2 Sensor platforms and embedded C . . . 6
1.3 TinyOS . . . 8
1.4 The CProver tools . . . 10

2 Verification for TOSThreads **14**
2.1 Overview . . . 14
2.2 Modelling the TinyOS kernel . . . 15
2.3 Categories of errors in context-aware software . . . 17
2.4 Verification and results . . . 18

3 Verification for MSP430 Embedded C **22**
3.1 Overview . . . 22
3.2 TOS2CProver: source-to-source transformation . . . 23
3.3 TOS2CProver: IRQ instrumentation . . . 23
3.4 Instrumentation with assertions, nondeterminism and assumptions . . . 27
3.5 CBMC and bounded program unwinding . . . 28
3.6 Verification and results . . . 29

4 Related Work **36**
4.1 Runtime safety . . . 36
4.2 Hybrid approaches: safety by emulation . . . 37
4.3 Verification and simulation . . . 38

Bibliography **41**

List of Figures

1.1 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. 5
1.2 MSP430 F1611. Selected I/O pins and their connections on TelosB motes. 7
1.3 (left) Memory organization for MSP430 microcontrollers. (right) Individual memory addresses for selected peripheral ports and MCU-internal registers. 8
1.4 Overview of a typical TinyOS nesC application. Modules, configurations and interfaces. 9
1.5 Sample wiring for a typical nesC application 9
1.6 CBMC program transformation into a mathematical model 11
1.7 The Counterexample-Guided Abstraction Refinement technique 13

2.1 A TOSThreads application's schematic control flow graph, calling TinyOS kernel functions, as on a Tmote Sky mote with integrated sensors for temperature, humidity and light intensity. 15
2.2 Overview and summarized error traces for the *PatientNode* application . . 20

3.1 The verification tool chain 22
3.2 (a) Naive refactoring the application to emulate hardware interrupts by running IRQ handlers concurrently with the main program line. Solid lines denote atomic code; 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 is now de facto atomic. After a further reduction step, this program is passed to CBMC. 25
3.3 Verification times for selected memory-violation assertions in *Sense* 34

List of Tables

2.1 Exemplary models for the radio driver 16
2.2 Selected CProver models of TOSThreads' synchronization API 17
2.3 Selected sensor and LED driver functions 18
2.4 Categories of errors in context-aware, TinyOS applications 18
2.5 Categories of errors in generic concurrent software 18
2.6 Verification benchmarks. *Blink*, a basic LED-actuating application, is included for reference. *SenseAndSend* monitors on-board sensors; *PatientNode* is a SenseAndSend extension for distributed monitoring. . . . 19

3.1 `tos2cprover`: source-to-source transformation examples for MSP430 code. 24
3.2 The `ADC` IRQ instrumentation 25
3.3 TelosB-based test cases 29
3.4 Fragment of verification condition for *Sense* 35

4.1 Runtime diagnosis and recovery solutions to software errors 36
4.2 Sensor platform emulators 38

Chapter 1

Introduction and Background

1.1 Introduction

While small applications for basic embedded systems centered around 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 becomes 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, and a multihop version of the same application into over 25×10^3 lines; similar program sizes are yielded from TelosB[2] MSP430 `elf` files. Not even programming embedded C (for the platforms' own C compilers, e.g., `avr-gcc` and `msp430-gcc` [39]) decreases the code's complexity—the application will have roughly the same size[3].

Instead of programming sensor applications from scratch, *operating systems* have been developed for sensor nodes, such as TinyOS [27] and Contiki [37]; these OSs predefine

1. scheduling policies, and
2. driver code for interfacing with the hardware.

As a result, most programmers only need write application logic in a high-level language, while calling scheduling and driver functionality from the operating system's existing code base. In order to then deploy the resulting application on a particular

[1]Mica motes are based around the Atmel AVR ATmega128L 8-bit microcontroller [3].

[2]Telos platforms (e.g., TelosB [31]) are built with MSP430 [36], 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 [27] operating system. Thus, optimizations over these numbers may be feasible.

mote platform, the operating system's compiler will generate a platform-specific embedded C program by translating and inlining the programmer's application with all the OS code base that the application references, and then passing that to the platform's C compiler, which in turn generates machine code.

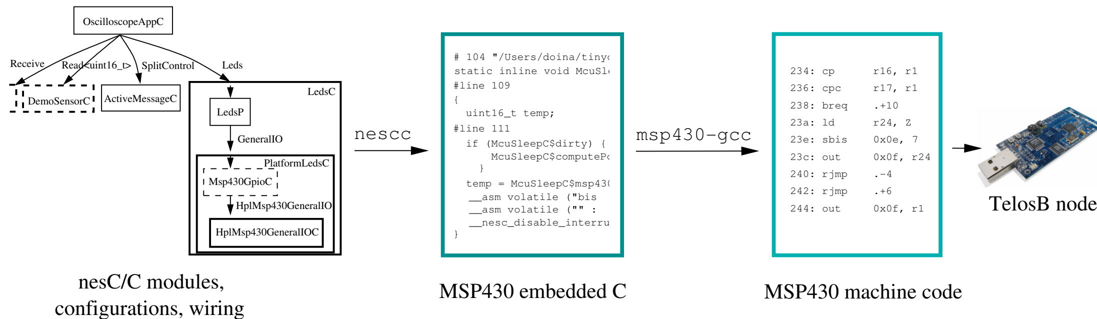


Figure 1.1: TinyOS program compilation for a TelosB mote. The nesC compiler, `nescc`, inlines nesC/C components into MSP430 embedded C; `msp430-gcc` then outputs machine code.

As an illustrative example, Fig. 1.1 overviews TinyOS's tool chain of program compilation for a TelosB mote. The *Oscilloscope* application is a nesC [18] (or TOSThreads [22]) 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 MSP430 embedded 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. 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.,:

- infinite-state programs cannot in general be guaranteed verification,
- complex data structures are rarely supported by existing software verifiers,
- even finite-state programs, particularly multithreaded ones, give rise to too large a set of program states to be verified within certain time and memory limits, and
- verifying against *safety* specifications (i.e., a statement of the form "A never happens") generally scales better than against *liveness* specifications (i.e., "B eventually happens").

Then, as overviewed in Fig. 1.1, verifying sensor software can mean verifying either (i) OS-specific code such as TinyOS components, or (ii) OS-less, but platform-specific

embedded C, or, finally (iii) OS-less, platform-specific machine code. In the subsequent two chapters, this report describes verification methods and tools for the first two such approaches.

Chapter 2 describes a software verification tool for platform-independent, OS-dependent sensor programs: multithreaded TinyOS applications written over the C TOSThreads library. This approach of verifying one module at a time naturally decreases the program's state space, such that—even as the application is multithreaded—safety specifications over fairly complex monitoring applications can be verified in under one hour's time. In order to automatically extract a correct model out of the single TOSThreads module to be verified, we need to manually construct brief models of those OS driver modules called by the application. The resulting program is then passed to SatAbs [8], a software verifier for multithreaded ANSI C, with specifications (written as C assertions) pinpointing incorrect interface use and incorrect adaptation to environmental input.

In turn, chapter 3 describes platform-dependent, OS-independent orthogonal approach: a software verification tool for large, OS-wide programs written in MSP430 embedded C with asynchronous hardware interrupts. Our tool automatically translates the program into standard C by replacing direct memory access with a model of the MCU's memory map. A number of calls to hardware interrupt handlers are inserted into the main application to emulate the existence of hardware, and their occurrence is minimized with a partial-order reduction technique, in order to decrease the program's state space. Safety specifications are written as C assertions embedded in the code. The resulting sequential program is then passed to CBMC [7], a bounded software verifier for sequential ANSI C. Besides memory-related errors (e.g., out-of-bounds arrays, null-pointer dereferences), this tool chain verifies application-specific assertions, including low-level assertions upon the state of the registers and peripherals.

In the remaining of this chapter, we give background information essential to either approach. Section 1.2 describes the syntax and semantics of MSP430 embedded C, by briefly overviewing the MSP430 microcontroller, its peripherals and memory map, and its use on the TelosB platform. Section 1.3 does the same for TinyOS-specific, nesC and TOSThreads code. Sections 1.4.2 and 1.4.1 overview SatAbs and CBMC, the two software verifiers for ANSI C that we base on.

1.2 Sensor platforms and embedded C

A variety of hardware platforms are available as sensor nodes. TelosB [31] motes are based on the 16-bit Texas Instruments MSP430 [36] microcontroller; Mica nodes are built around Atmel's AVR [3], and MITes nodes around Intel's 8051 [2]. The MSP430 is 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, 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. 1.2).

Pins such as those supplying voltage (1 and 62-64) or the two crystal oscillators'

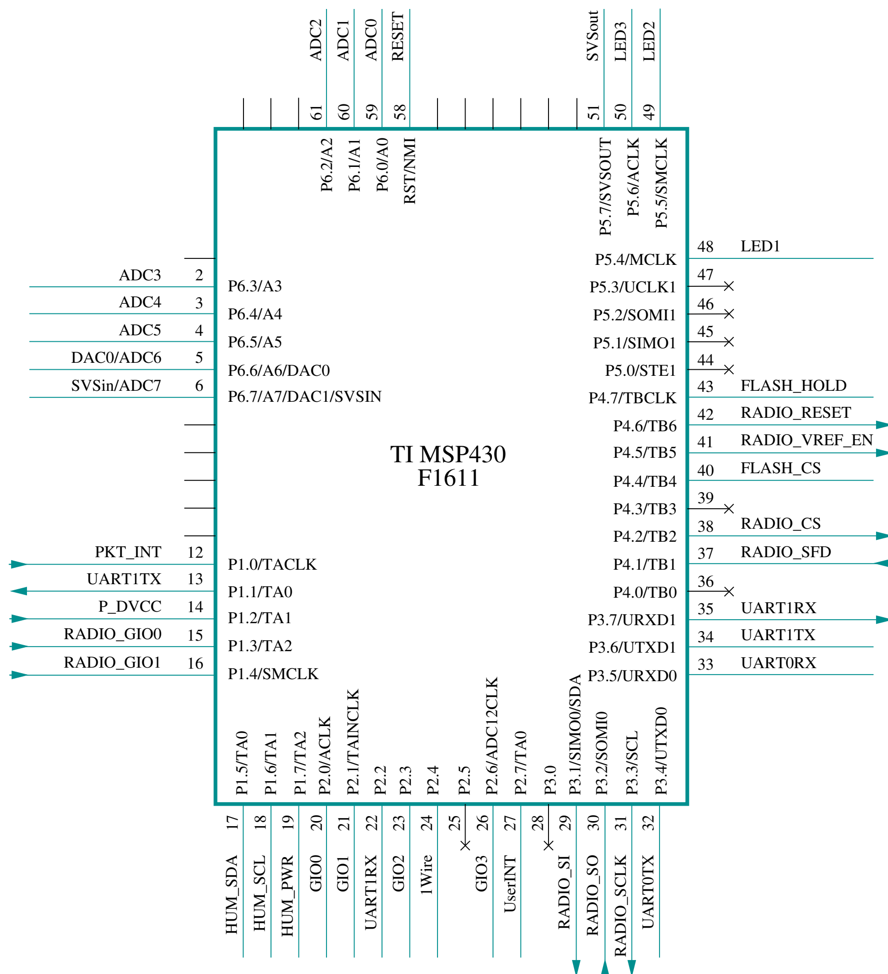
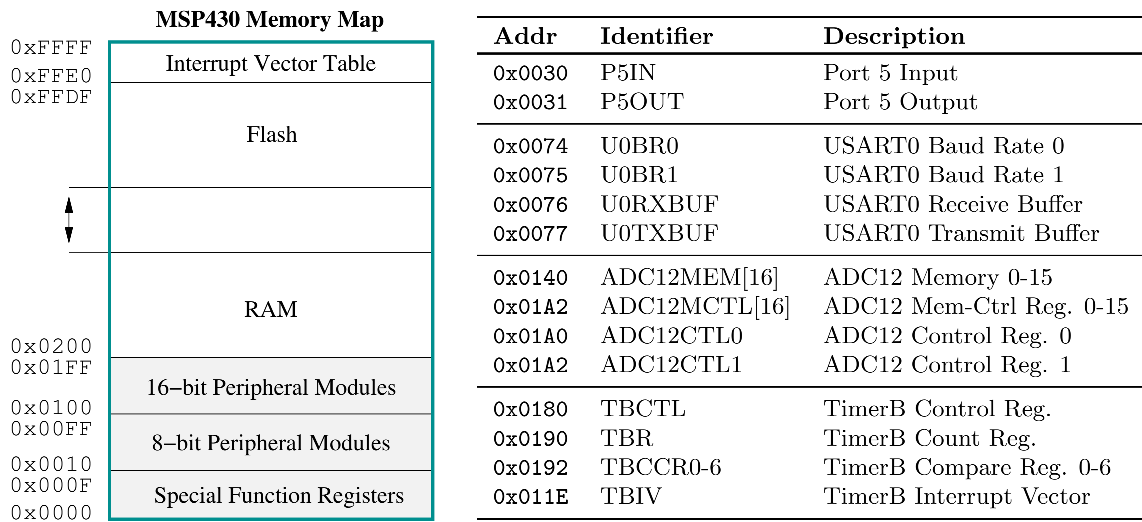


Figure 1.2: MSP430 F1611. Selected I/O pins and their connections on TelosB motes.

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, this pin is connected to the bidirectional serial data port, HUM_SDA, of the on-board humidity sensor (which produces a digital output). Similarly, pins 32-33 are transmit and 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.

Any embedded C software would be able to access these pins, together with other peripheral modules and registers, by direct memory access to memory addresses—as dictated by the MCU's memory mapping overviewed in Fig. 1.3 (left). All on-board memory, including peripherals and the Interrupt Vector Table, are mapped into a unique



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 Mem-Ctrl Reg. 0-15
0x01A0	ADC12CTL0	ADC12 Control Reg. 0
0x01A2	ADC12CTL1	ADC12 Control Reg. 1
0x0180	TBCTL	TimerB Control Reg.
0x0190	TBR	TimerB Count Reg.
0x0192	TBCCR0-6	TimerB Compare Reg. 0-6
0x011E	TBIV	TimerB Interrupt Vector

Figure 1.3: (left) Memory organization for MSP430 microcontrollers. (right) Individual memory addresses for selected peripheral ports and MCU-internal registers.

address space, with Special Function Registers and peripheral modules at low addresses.

The MCU's six 8-bit I/O ports whose pins are overviewed in Fig. 1.2 are mapped in the `0x10-0xFF` address space for 8-bit peripheral modules. The 8-bit output register for port 5 is accessed as `0x0031`; setting bits 4-6 in this register controls the LEDs. Thus, when code in this MCU's specific `msp430-gcc` [39] embedded C writes:

```
static volatile uint8_t r __asm ("0x0031");
r &= ~(1 << 6);
```

or in other words

```
*(volatile uint8_t *)49U |= ~(1 << 6);
```

this amounts to setting a bit in the 8-bit output register `P5OUT` of peripheral port 5 at location `0x0031`, where LEDs are accessed, which turns the yellow LED on.

Other essential mapping examples are shown in Fig. 1.3 (right).

Similarly, a function declared with the attributes

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

declares that `sig_ADC_VECTOR` is an interrupt service routine for interrupt line 14, and that it wakes the processor from any low power state as the routine exits.

1.3 TinyOS

TinyOS (in the form of its two major releases, TinyOS 1 [20] and 2 [27]) is the mainstream operating system for wireless sensor network devices. The operating systems itself, as well as most programmers' applications, are implemented in TinyOS's own *nesC*

(network embedded systems C) language [18]. NesC software comes in *components*, either *modules* or *configuration* (Figs. 1.4 and 1.5 give an 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* [17]. Given the language's focus on embedded systems, an application's components and interfaces are settled at compile time, unlike in the case of objects, and for efficiency.

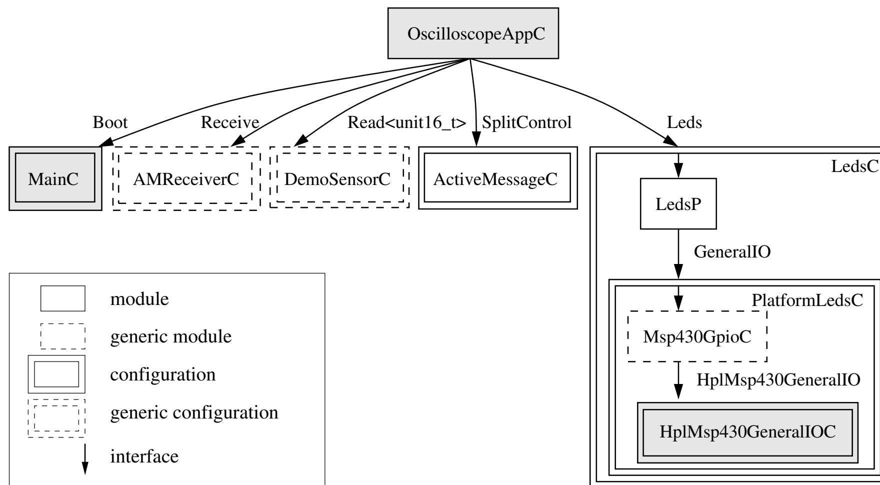


Figure 1.4: Overview of a typical TinyOS nesC application. Modules, configurations and interfaces.

```
module OscilloscopeAppC {
  uses {
    interface Leds;
    interface Boot;
    interface Receive;
    interface AMSend;
    interface Timer<TMilli> as MilliTimer;
    interface Packet;
    interface Read<uint16_t>;
    interface SplitControl as RadioControl;
  }
}
```

```
configuration LedsC {
  provides interface Leds;
}
implementation {
  components LedsP, PlatformLedsC;

  Leds = LedsP;

  LedsP.Init <- PlatformLedsC.Init;
  LedsP.Led0 -> PlatformLedsC.Led0;
  LedsP.Led1 -> PlatformLedsC.Led1;
  LedsP.Led2 -> PlatformLedsC.Led2;
}
```

Figure 1.5: Sample wiring for a typical nesC application

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 preempt the lower-priority tasks, which

execute most of the program logic. Tasks run to completion, 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 both synchronous and 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 designed under the expectation that a final, inlined code will be generated from components by whole-program compilers, a fact which allows for better code generation and analysis.

TOSThreads [22] is TinyOS 2.x's recent thread library. It preserves the system's existing concurrency model, and adds a lowest third priority level for the new, *application threads*. The threads are now allowed blocking system calls, which execute as regular TinyOS tasks, and which unblock the application thread when completed; these system calls are implemented as an abstraction on top of regular nesC interfaces. As such, TOSThreads provide the traditional, POSIX-threads-like programming paradigm for TinyOS, can be programmed in either nesC or C, and come complete with synchronization methods.

1.4 The CProver tools

CBMC [7] (Section 1.4.1) and SatAbs [8] (Section 1.4.2) are part of (and share the frontend of) the CProver [24] tool suite for checking ANSI-C programs. Both tools take program specification written as *C assertion statements*, and are geared roughly towards verifying embedded systems programming, which is—more so than high-level applications—both critical and a challenge to debug, due to the extensive use of pointers, pointer arithmetic, and bit-wise operators. The tools' input language is standard C, and, unlike other C model checkers, they support 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].

This allows the tools to pinpoint program errors related to bit-level operators and arithmetic overflow, besides other errors concerning pointer and array operations, arithmetic exceptions, user-inserted assertions and assumptions. Assertions are input either as standard `assert(__radio_on);`, or as the CProver-specific statement:

```
__CPROVER_assert(__radio_on, "Radio send w/o radio start");
```

which allows the string message to identify the assertion, for debugging purposes. An assumption states a boolean condition, which discards from checking any program trace which doesn't satisfy it (and, thus, may be used to model synchronization procedures):

```
__CPROVER_assume(!mutex->lock);
```

To derive an accurate mathematical representation of an input program, both tools first translate the C input into a form of `goto-cc`, a side-effect-free intermediate represen-

[4]This fact amounts to some issues for embedded C code which relies on, e.g., the memory alignment of `struct` fields.

tation of ANSI-C, and apply different further transformations and checking procedures, as follows.

1.4.1 CBMC—A bounded software verifier for ANSI C

CBMC reduces the problem of checking an infinite-state, sequential ANSI-C program to the satisfiability of a boolean formula. It implements bounded model checking [9]; this unwinds the program and its specification to obtain a boolean formula that is satisfiable if there exists an error trace. The formula is then checked by using a SAT procedure. If the formula is satisfiable, a counterexample is extracted from the output of the SAT procedure.

For that, an initial program transformation phase does so that:

1. All side-effect assignments are broken into equivalent statements by introducing auxiliary variables, and all loops (`for, while`, backward `gotos`) and recursive function calls are *unwound* a user-provided number of times, by multiplying the loop body. For original loops which were guarded by a condition, each copy of the loop is in turn guarded by an `if` statement that uses the original loop condition.

2. Function calls are inlined (with any `return` statement replaced by an assignment to a fresh variable), and recursive such calls are unwound similarly.

The resulting program consists of `if` instructions, assignments, assertions, labels, and forward jumps (`goto` instructions with subsequent jump targets). Variables may be nondeterministic, simply by being assigned the return value of an unimplemented function, e.g., `x=nondet();`, which is transformed at inlining time into a nondeterministic value.

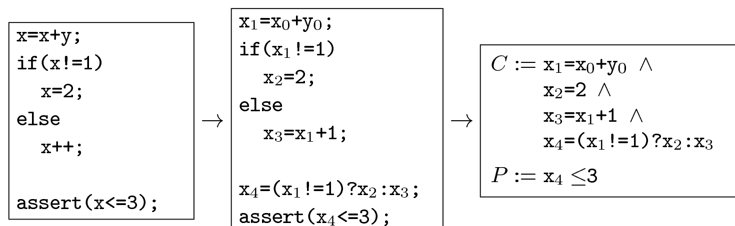


Figure 1.6: CBMC program transformation into a mathematical model

This program is then translated (using a pointer analysis in the process) 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 (as in Fig. 1.6, center). Frequently used for compiler optimizations, the technique simplifies the analysis of the variables' definition and use.

Two boolean propositional formulas are derived from the program in SSA form: C for the program itself, and P for the asserted expression, as in Fig. 1.6, 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 [25]. This transformation is word-width adjustable to, e.g., 16 bits, to simulate different hardware platforms. *Uninterpreted functions* model, e.g., `malloc` operations, which are too expensive to represent in bit-accurate form.

P is then verified by converting $C \wedge \neg P$ into conjunctive normal form (CNF) and then passing it to a SAT solver such as MiniSAT [35]. 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.

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), is highly automated and scales reasonably well.

1.4.2 SATABS—A software verifier for multithreaded ANSI C

SatAbs reduces the problem of checking an infinite-state, concurrent ANSI-C program to that of a sound, but potentially non-terminating sequence of solving satisfiability problems. It implements *predicate abstraction* to reduce the program's state space: Fresh, (*abstract*) boolean variables are determined so that they replace *boolean relations* of the original variables; e.g., a boolean predicate `b1` may replace the boolean expression `barrier.count > 0`, in which `barrier` is an implementation of a synchronization barrier.

A *concrete transition relation* is the direct translation of the input C program in SSA form into a state machine, in which states are valuations of program variables, and transitions are program flow control relations. An *abstract* model is then derived from the single-assignment, inlined-functions version of the input model by calculating a new *abstract transition relation* in which a transition exists between two abstract states only if there exist abstractions of two concrete states, between which the concrete model has a transition. Similarly, concrete program specifications, inputted in the form of C assertions, are boolean conditions over the variables in the SSA program; their abstract translation are then boolean conditions over abstract variables.

The abstract model is then a boolean program which registers as a much smaller model (small enough for checking by a BDD-based model checker), with data detail removed. The predicates are chosen with the view that detail relevant to the property to be verified is preserved.

This abstraction conserves reachability properties: if a property is proven to hold for the abstract model, it will also hold on the original program. On the other hand, no conclusion can be drawn from the opposite situation, in which the checking of the abstract program fails, the abstract *counterexample* demonstrating failure (i.e., a transition trace leading to the violation of an abstract specification) may not have a corresponding concrete counterexample (and is, thus, called "spurious").

An automated *Counterexample Guided Abstraction Refinement* (CEGAR) proce-

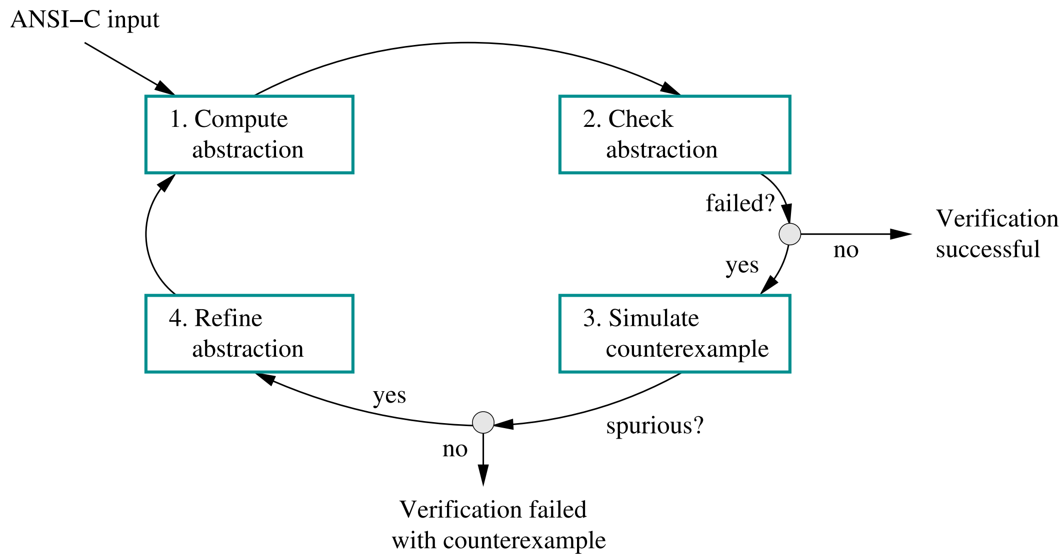


Figure 1.7: The Counterexample-Guided Abstraction Refinement technique

dure (overviewed in Fig. 1.7) caters for such cases. The boolean program that is the abstract model is passed to a model checker (of which SatAbs supports a variety, such as SMV [8]); if this model checker returns a counterexample over a failed assertion, this may be spurious and needs simulating on the concrete model. If this simulation fails, the abstract model is refined by adding further abstract predicates using weakest preconditions. Each such abstraction iteration only adds minor changes to the abstract model, which makes the use of incremental model checking desirable.

The CEGAR technique is *sound* (i.e., it never gives a false report, be it a "Verification successful" or a "Verification failed" with counterexample); it is also *complete* for finite-state input programs, but incomplete otherwise, with no termination guarantee given.

Chapter 2

Verification for TOSThreads

2.1 Overview

To prevent the occurrence of context-awareness and concurrency bugs in sensor software *before* deployment time, we provide the first verification tool for multithreaded TinyOS 2.x [27] applications written in TinyOS's synchronous C TOSThreads API [22]. We are able to statically verify a TinyOS application against *safety specifications*, which require the program to be in a "safe" state with regard to its observable behaviour and memory state, given a—possibly nondeterministic—pattern of data incoming from the environment.

Our safety specifications are tailored to applications written for monitoring, single-hop *body sensor networks*. They may guard against standard programming errors such as failing to write exception-handling code in case of, e.g., an on-board sensor failing to provide a data reading when requested, or—similarly—an expected radio packet failing to materialize. With more effort put into writing specifications, more complex program behaviour can be pinpointed, such as the fact that a sensor node will issue a radio packet with only bounded time delay, even in the case of exceptions.

The verification method employs is *scalable*: We check a given application modularly, by extracting it from the rest of the TinyOS kernel, and replacing the latter with interface-preserving models. While this requires reviewing the TinyOS code base to learn the semantics of all system calls, the method is good value for developers: it only needs to be provided once, is reusable by all applications and (given that TinyOS is fairly stable) requires little maintenance over time. Furthermore, as sensor programs generally run in an infinite loops, we manually bound the number of loop iterations to gain a finite program.

Our tool (published as [4]) builds on SatAbs [8], a generic software verification tool for multithreaded ANSI C; as described in Section 1.4.2, SatAbs takes specifications written as user-specified assertions and assumptions 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 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) write safety specifications as sets of C assertions and assumptions, and subsequently verify application and kernel model. We report benchmarks on running our tool on staple programs: a standard monitoring application distributed with TinyOS's sources, and on a more complex healthcare application; we find basic, routine violations of safety requirements.

In what follows, we present our verification method. We first overview multithreaded TinyOS applications and our modelling away of the OS kernel, then underline possible sources for software bugs. Finally, we assess performance with a set of benchmarks and point to the cause and nature of the bugs found.

2.2 Modelling the TinyOS kernel

A TOSThreads application, as depicted in Fig. 2.1, is tightly connected to the rest of the operating system's kernel by calling existing OS kernel functions (as described in Section 1.3); these either manage execution scheduling (e.g., thread management) or are driver code for, e.g., the radio port, on-board sensor chips or LEDs.

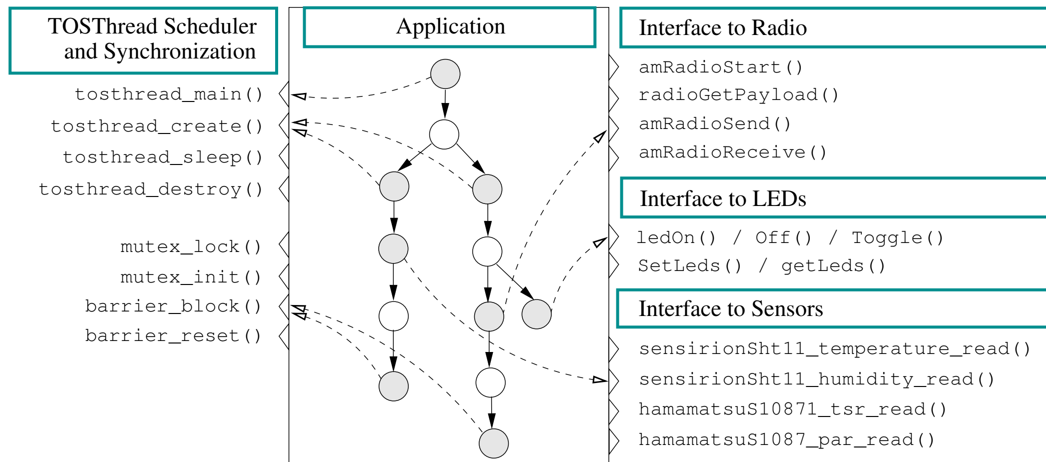


Figure 2.1: A TOSThreads application's schematic control flow graph, calling TinyOS kernel functions, as on a Tmote Sky mote with integrated sensors for temperature, humidity and light intensity.

When analysing all possible execution traces of a program to check for violations of a specification, the complexity of the analysis increases with the number of program states; when the program is concurrent, the complexity increase is exponential,

due to exploring all possible context switches between threads. To minimise the state space under analysis, we verify an application individually: instead of analysing the application code together with the sizable, platform-dependent implementation of the many kernel functions (referred in Fig. 2.1), we write new, brief implementations for these functions, ensuring that their interface's exceptional behaviour is preserved; e.g., if `amRadioSend([..])` can fail returning an error code, so can its model. We overview selected such function models in what follows.

Table 2.1: Exemplary models for the radio driver

```
error_t amRadioStart()
{
  success = nondet_bool();
  busy = nondet_bool();

  if(success) {
    __radio_on = true;
    return SUCCESS;
  }

  if(busy)
    return EBUSY;

  return FAIL;
}
```

```
error_t amRadioSend(am_addr_t addr,
  message_t* msg, uint8_t len, am_id_t amId)
{
  success = nondet_bool();
  busy = nondet_bool();

  __CPROVER_assert(__radio_on,
    "Radio send w/o radio start");

  if(success) {
    __sent_msg = *msg;
    __sent_am  = amId;
    __time    += MIN_HW_DELAY;
  }
  ...
  if(busy)
    return EBUSY;

  return FAIL;
}
```

Table 2.1 shows sample modelling code of two functions from the radio driver, as listed in Fig. 2.1 (with some lines and data types removed for legibility). A new boolean variable `__radio_on` is invented to model the state of the radio interface, initialized with `false`, and set or reset, respectively, by `amRadioStart()` (Table 2.1, left) and `amRadioStop()`. `amRadioStart` nondeterministically returns `SUCCESS` (and sets `__radio_on`), `EBUSY` or `FAIL`.

Similarly, the sensor node is given a model of the wireless communication medium; sending and receiving messages become assignments into node-local fresh variables `__sent_msg` (of the same type as a packet's payload) and `__sent_am` (an AM identifier[1]).

[1] In order to allow families of application or services to communicate on the same wireless channel, TinyOS implements the *Active Message* (AM) protocol layer to multiplex access to the radio. Each service should be assigned an AM identifier by the programmer, which functions like a supplementary address field. A network node:

- sends a packet with a particular value for its destination `AM` by passing that value in the `amId` argument of `amRadioSend(addr, msg, ..., amId)`;

Table 2.1 (right) overviews the model of amRadioSend, which includes an assertion upon the state of the radio interface, then (also nondeterministically) assigns __sent_msg and __sent_am, and increments another local variable modelling time. The model of amRadioReceive behaves similarly.

Table 2.2: Selected CProver models of TOSThreads' synchronization API

```
typedef struct mutex {
  bool lock;
} mutex_t;
void mutex_init(mutex_t* m)
{
  m->lock = false;
}
error_t mutex_lock(mutex_t* m)
{
  __CPROVER_HIDE:
  __CPROVER_assume(!m->lock);
  m->lock = true;
  return SUCCESS;
}
```

```
typedef struct barrier {
  uint8_t count;
} barrier_t;

void barrier_reset(barrier_t* b, uint8_t count)
{
  __CPROVER_HIDE:
  b->count = count;
}

void barrier_block(barrier_t* b)
{
  __CPROVER_HIDE:
  __CPROVER_assume(b->count > 0);
  b->count--;
  __CPROVER_assume(b->count == 0);
}
```

We easily match TOSThreads' thread-creating and scheduling functions to the POSIX API that CProver supports. Then, Table 2.2 gives part of our model of TOSThreads synchronization techniques, in CProver lingo, while Table 2.3 overviews sensor and LED driver models.

2.3 Categories of errors in context-aware software

In addition to generic memory violation issues, a concurrent context-aware program will also exhibit additional programming errors, which are the focus of our verification. We categorise these from two points of view: generic concurrency (as in Table 2.5) and correct awareness of context (Table 2.4). One particular bug can be seen from both viewpoints: an application blocked in waiting for a network packet from a network neighbour which unexpectedly moved away exhibits a bug categorized both as a *network exception* and as *deadlock*.

- may receive and unpack a message with amRadioReceive(addr, ..., amId) if both (i) the local node address TOS_NODE_ID matches the packet's destination address (or the latter is AM_BROADCAST_ADDR), and (ii) the AM identifier amId passed to amRadioReceive matches the AM field of the packet.

Table 2.3: Selected sensor and LED driver functions

```
error_t sensirionSht11_temperature_read
  (uint16_t* val)
{
  success = nondet_bool();
  busy = nondet_bool();

  if(success) {
    *val = nondet_uint16();
    return SUCCESS;
  }

  if(busy)
    return EBUSY;

  return FAIL;
}
```

```
bool __led0 = false;
bool __led1 = false;
bool __led2 = false;

void led0On()
{
  __CPROVER_HIDE:
  __led0 = true;
}

uint8_t getLeds()
{
  __CPROVER_HIDE:
  uint8_t mask = 0;

  mask |= __led0? 1 << 0: 0;
  mask |= __led1? 1 << 1: 0;
  mask |= __led2? 1 << 2: 0;

  return mask;
}
```

Table 2.4: Categories of errors in context-aware, TinyOS applications

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

Table 2.5: Categories of errors in generic concurrent software

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

2.4 Verification and results

We first look at *SenseAndSend*, a staple monitoring application in the TinyOS source tree. Four working threads monitor the Tmote Sky on-board sensors, and each write a fresh value in the data field of a network message; a fifth thread sends the message on the radio. The program's natural-language specification[2] states that such messages should be sent periodically, containing valid readings, and accompanied by LED signalling.

The *claims* are specifications written in assertion form.

[2]We recovered most specifications from the application's README file.

Table 2.6: Verification benchmarks. *Blink*, a basic LED-actuating application, is included for reference. *SenseAndSend* monitors on-board sensors; *PatientNode* is a SenseAndSend extension for distributed monitoring.

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

We give the results of our verification runs[3] in Table 2.6. LOC stands for Lines of Code in the application's control-flow graph, including the kernel model; whenever a bug exists, it is categorised as in Tables 2.4 and 2.5.

Claim 79 uncovers a misuse of the radio interface; thread `tosthread_main` fails to ensure that the radio is turned on by not checking `amRadioStart`'s returned error code, before a call to `amRadioSend`:

```
// error: amRadioStart may fail
amRadioStart();                                     // thread tosthread_main

if(amRadioSend(AM_BROADCAST_ADDR, &send_msg, ..) // thread collection_send_thread
```

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

```
sensor_data = radioGetPayload(&send_msg, [..]);  // thread tosthread_main

// error: sensirionSht11_temperature_read may fail
read_sensor(sensirionSht11_temperature_read,     // thread temperature_thread
            &(sensor_data->temp));

amRadioSend(AM_BROADCAST_ADDR, &send_msg, [..]); // thread collection_send_thread
```

For the verification of this claim, we made an adjustment to the original program code and initialized `sensor_data->temp` (and all other variables reading from the sensors) to a constant, "magic" value outside the expected range for sensed data; this can then be compared against at packet-sending time.

[3]Verification times are given for runs on a Mac OS X with a 2.4GHz Duo Intel Core and 2GB RAM.

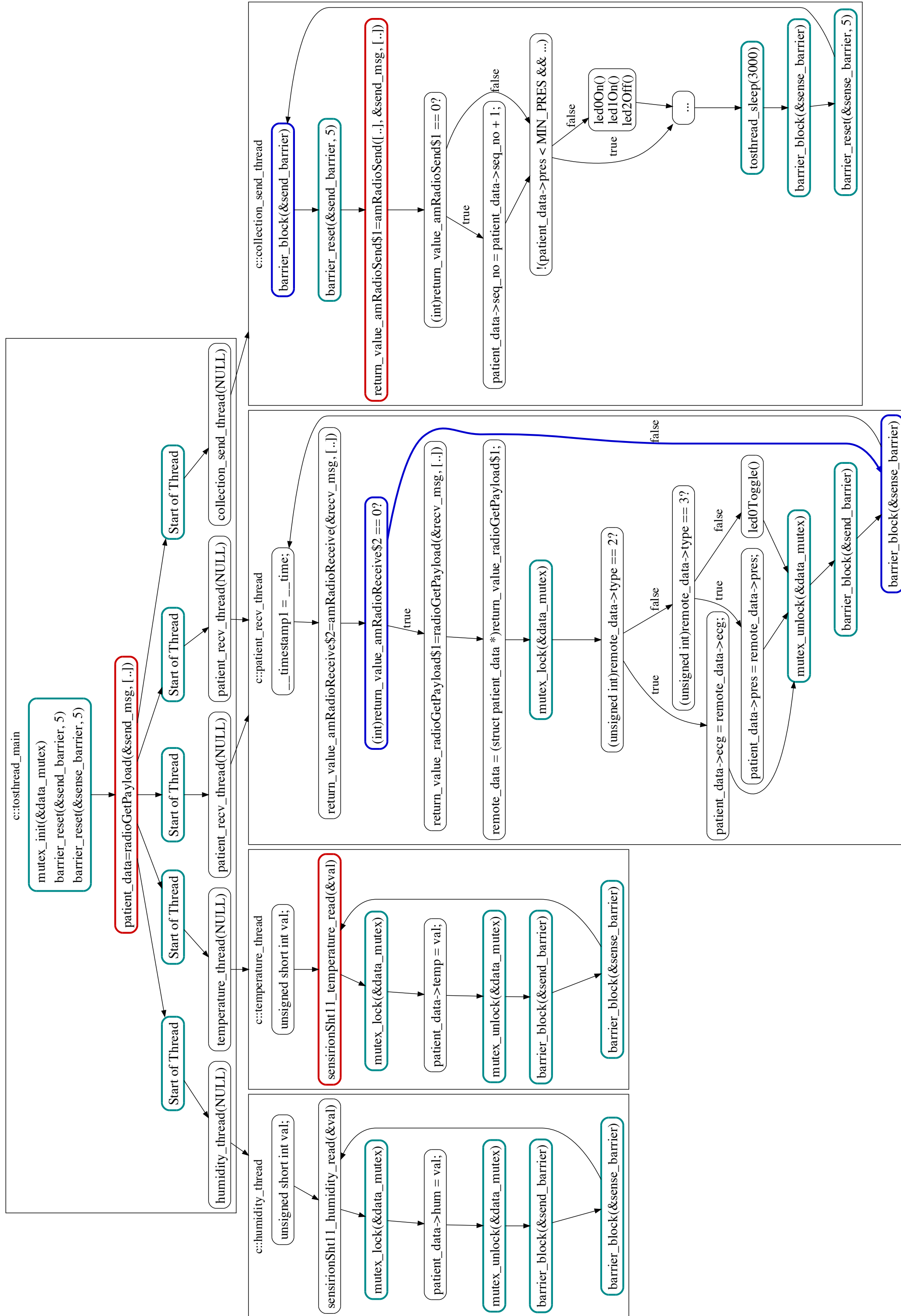


Figure 2.2: Overview and summarized error traces for the *PatientNode* application

PatientNode is a SenseAndSend extension tailored for monitoring patients in a pervasive healthcare wireless network, on the lines of the survey in [40]. A number of biosensors monitor each patient; a PatientNode application resident on one such sensor (the control-flow graph of which is overviewed in Fig. 2.2) collects readings from all of the patient's sensors, sends them in a network message, and signals an abnormal condition by a lit-LED configuration. Claims 172 and 254 are re-verifications of SenseAndSend's corrected claims 79 and 136. In Fig. 2.2, the red program states summarize the error trace for SenseAndSend's claim 136, as listed above; the complete error trace sees the sensing and receiving threads each complete an iteration.

Claim 230 uncovers that a misplaced closing brace in `patient_recv_thread` brings the program into a deadlock on a barrier, if a message expected to be received doesn't show up:

```
if(amRadioReceive(&recv_msg, [..]) == SUCCESS) { // thread patient_recv_thread
    [..]
    barrier_block(&send_barrier);
} // error: this should precede the line above

barrier_block(&send_barrier);
```

(or, the program trace summarized by the blue program states and "false" transition in Fig. 2.2). Unless the transition is corrected to allow blocking on `send_barrier` regardless of the packet reception outcome, the thread `collection_send_thread` blocks after the thread's first statement.

Claims 268 and 262 verify application logic: the first, that an abnormal received reading is treated as a false alarm if it is not confirmed by a subsequent reception, and the second, that the maximum span of time between outgoing packets is bounded, regardless of the contents of incoming packets.

Chapter 3

Verification for MSP430 Embedded C

3.1 Overview

In TinyOS, software bugs stem from both the legacy nesC component base (the lowest levels of which are platform-dependent), and the programmer application's components. By a *safe* TinyOS program, we understand that which exhibits no memory violations, and whose programmer-inserted assertions hold, if reachable.

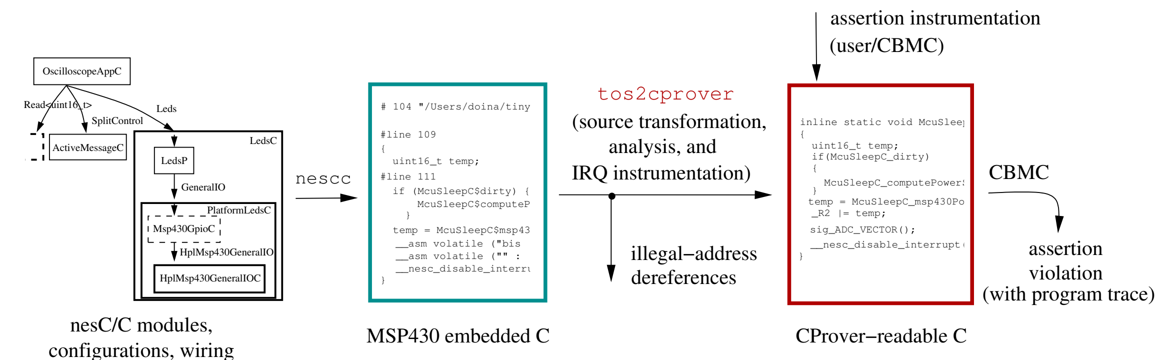


Figure 3.1: The verification tool chain

To achieve a homogeneous verification scheme for both legacy and newly programmed TinyOS code, we created in [5] an automated tool chain of program transformation and verification as depicted in Fig. 3.1. In order to be able to treat both nesC components and the C/nesC TOSThreads identically, an initial run of TinyOS's `nescc` compiler is used to generate an inlined, platform-specific, low-level C program; 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 has a double 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 takes over the functionality of a hardware emulator. A partial-order reduction (POR) technique is used to minimize the occurrence of such calls.

Following these transformation and instrumentation steps, the result is a high-level, standard C program which precisely (yet minimally) preserves the functionality of the initial platform-specific program and its 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 what follows, we detail `tos2cprover`'s source-transformation step in Section 3.2, its instrumentation of the source with calls to IRQ handlers (and the related POR technique) in Section 3.3, the various possibilities for instrumenting the program with assertions, assumptions and nondeterminism in Section 3.4, and the setting of bounds for program's unwinding with CBMC in Section 3.5. Finally, Section 3.6 gives verification results and runtime.

3.2 TOS2CProver: source-to-source transformation

Table 3.1 exemplifies the source transformations executed by `tos2cprover` on a TelosB, MSP430 program. While `mspgcc` code implicitly assumes an underlying memory map (overviewed in Section 1.2) in which low, constant addresses have a semantics (e.g., writing at `0x0031` programs the LEDs), this memory is replaced with a header file defining global replacement variables; e.g., `_P5OUT` is now the 8-bit output register for peripheral port 5. All subsequent dereferences of address `0x0031` are replaced into accesses to `_P5OUT`[1]. As a note, the Status Register `_R2` has the General Interrupt Enable (GIE) as bit 4; if GIE is set, interrupts are enabled.

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

3.3 TOS2CProver: IRQ instrumentation

The `nescc`-generated program inputted to `tos2cprover` doesn't explicitly call any IRQ handlers; in deployments, the calls are made from the hardware. Instead, it defines the

[1] Note that the variables we introduce are names in accordance to the MSP430 documentation [36], but are preceded by an underscore, to avoid name clashes with existing program variables.

Table 3.1: `tos2cprover`: source-to-source transformation examples for MSP430 code.

<table>
<tr><th>MSP430-specific program feature</th><th>Example <code>tos2cprover</code> transformation</th><th></th></tr>
<tr><td>MCU registers and memory map</td><td>Standard C global variables
<code>unsigned short _R2;</code>
<code>unsigned char _P5OUT;</code>
<code>unsigned short _ADC12CTL0;</code>
<code>unsigned short _ADC12MEM[16];</code></td><td>
(Status Register)
(<code>0x0031 P5OUT</code>)
(<code>0x01A0</code>)
(<code>0x0140</code>)</td></tr>
<tr><td>Fixed-address dereference
<code>*(uint8_t*)49U &= ~(0x01 << 6);</code></td><td>Global variable access
<code>_P5OUT &= ~(0x01 << 6);</code></td><td></td></tr>
<tr><td>Fixedly allocated variables
<code>uint16_t HplAdc12P$ADC12CTL0</code>
<code>__asm ("0x01A0");</code>
<code>HplAdc12P$ADC12CTL0 |= 0x0010;</code></td><td>Global variable access

<code>_ADC12CTL0 |= 0x0010;</code></td><td></td></tr>
<tr><td>Assembly instructions
<code>__asm volatile ("eint");</code>
<code>__asm volatile</code>
<code>("bis %0, r2": :"m"(temp));</code></td><td>C instructions
<code>_R2 &= 0x0008;</code>

<code>_R2 |= temp;</code></td><td></td></tr>
</table>

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:

```
void sig_TIMERB1_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[2], 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`.

To simulate the presence of interrupts, `tos2cprover` needs to instrument the program with explicit, atomic 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 (as TinyOS events always run to completion). A listing is given in Table 3.2.

A correct, yet naive, approach is to instrument the program by refactoring it to use threads, and running the IRQ instrumentation as separate threads alongside a `main` thread, as in Fig. 3.2(a). The 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

[2] As a note, with NesC components being statically wired through interfaces, the result is that a number of those functions in the `app.c` file generated by `nescc` which implement this wiring usually consist of a one-line function call.

Table 3.2: The ADC IRQ instrumentation

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

rooted in the call, we employ two automated minimization procedures to reduce the number of these sequential instrumentations, as described in the following.

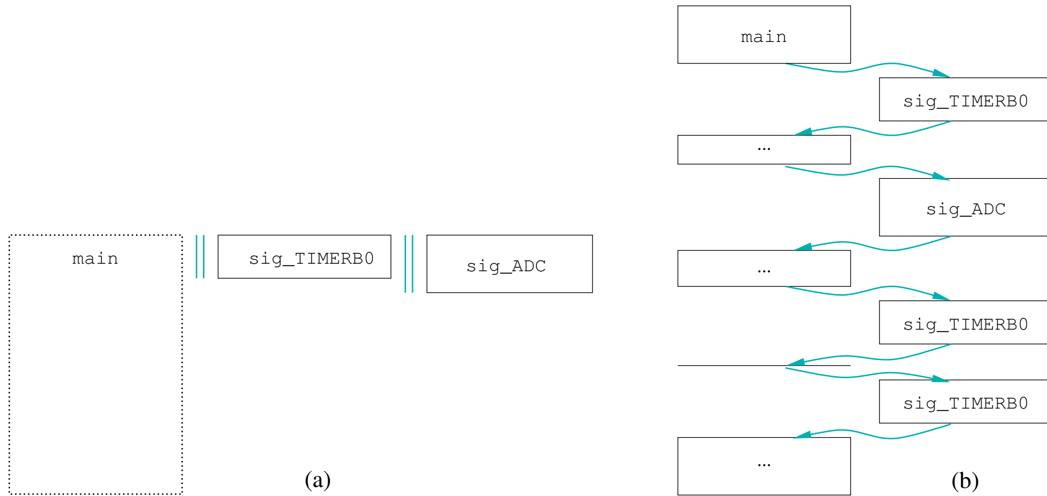


Figure 3.2: (a) Naive refactoring the application to emulate hardware interrupts by running IRQ handlers concurrently with the main program line. Solid lines denote atomic code; TinyOS events interrupt synchronous code and run to completion. (b) Efficient hardware emulation by partial-order reduction, resulting in a sequential program of reduced state space; all code is now de facto atomic. After a further reduction step, this program is passed to CBMC.

3.3.1 Decreasing the state space with a partial-order reduction technique

The first is a *partial order reduction* (POR) technique (or *model checking using representatives*) [10], a general method to reduce the state space of a concurrent program to be model checked. Applied on the original TinyOS 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 a single interleaving

representative needs checking. An image of the transformation of the program as a result of the POR technique is given in Fig. 3.2(b).

Formally, we take the concurrent refactoring of the TinyOS program as in Fig. 3.2(a) and give it a standard formalization as follows. A system *state* $s \in S$ is a valuation of all program variables before or after a C statement[3]. The *transition* set T contains state tuples, $T \subseteq S \times S$; we write $\alpha \in T$ or $s \stackrel{\alpha}{\to} s'$ for a single transition; a *path* π is a sequence of transitions. Transitions intuitively model C statements and the program's control flow: when executed from a state s, a transition α leads the program into a new state s'. A transition α from T is called *enabled* in state s if $\exists s' \in S.s \to s'$, with the set of transitions enabled in s denoted by $enabled(s)$. Then, the concurrent program's equivalent Kripke structure is $M = (S, T, \{s_0\})$, with s_0 the unique initial state of the program. This we call the *full-state graph*, as T models all possible interleavings between the program's threads.

As standard, 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 should satisfy a soundness condition: When $ample(s)$ replaces $enabled(s)$, the soundness of the model checking algorithm must be preserved.

The basis for soundly calculating $ample(s)$ relies on an *independence* relation I between transitions, and on the *invisibility* of a transition in regard to a property to be verified [10]. The *independence* relation $I \subseteq T \times T$ is a symmetric relation satisfying, for each $s \in S$ and for each $(\alpha, \beta) \in I$, 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 $\alpha(\beta(s)) = \beta(\alpha(s))$.

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

In turn, the *invisibility* of a transition is defined such that $s \xrightarrow{\alpha} s'$ is *invisible* with regard to a property to be verified if executing the transition does not change the truth value of the property from state s to s'. Since in our case a property is a boolean expression over variable valuations, the definition effectively means that α is invisible iff, in what regards the program variables *in scope*, s is equal to s'.

With these definitions, the sound reduction from $enabled(s)$ to $ample(s)$ must satisfy the following constraints upon $ample(s)$ [10]:

C_0 $ample(s) = \emptyset$ iff $enabled(s) = \emptyset$.

[3]Our 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.

C_1 Along every path of the full-state graph starting in s, a transition dependant on a transition α from $ample(s)$ must be preceded by α. This effectively allows deferring transitions and is the .

C_2 Every $\alpha \in ample(s)$ is invisible, if s has had enabled transitions removed.

C_3 Within a cycle, if a certain transition is α enabled in at least a state, α must be included in an ample set on the cycle.

This first POR-based minimization step leaves, e.g., in the case of *Sense*, a total number of 91 IRQ calls between the two types of IRQs.

3.3.2 Improving POR's overapproximated race criterion by reachability checks

A final analysis step is used to further minimize the instrumentation count described in Section 3.3.1. To remove some of the overapproximation degree, `tos2cprover` inputs the instrumented program to CBMC for a preliminary run which only checks the *reachability* of each 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 *Sense* with a manageable set of 8 reachable IRQ calls; also, this CBMC run is timewise 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.

3.4 Instrumentation with assertions, nondeterminism and assumptions

Our tool chain verifies, for each inputted program, two types of assertions. For both the existing TinyOS code base and any new applications, assertions can 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 variable local to a component).

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. 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[4]. Advantageously, this generation is completely automatic, with CBMC analysing an array's declaration to find the index bounds; SafeTinyOS [11], for example, has programmers explicitly type-annotate arrays with access bounds.

Finally, a decision needs to be taken in 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 TinyOS 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 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 envolved 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, particular set.

3.5 CBMC and bounded program unwinding

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 at a time, for scalability. The runs need to have specified an unwinding depth; this can be either the 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 the 16 loops from *Sense*, the loop:

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

is always bounded at 16 iterations (the size of the ADC12 conversion memory)[5]. Other loops, on the other hand, are clearly unbounded, such as the main OS scheduler loop in

[4]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 correspondent in the original `nesc` code.

[5]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.

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 depths for the rest. For the purpose of determining the reachability of instrumented IRQ calls (described in Section 3.3), we set the depth for the unbounded loops to the minimum, 1.

3.6 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 3.3.

Table 3.3: TelosB-based test cases

	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 8	initial 422, minimized to 30

We detail the size and complexity of the test cases in terms of (i) the 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. As a side note, the program generated by nescc and inputted to tos2cprover is not completely 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.

Finally, Table 3.3 gives the precise number of IRQ instrumentations calculated as in Section 3.3, 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 remaining of this section, we give an overview of our verification runs, discuss their scalability, advantages and limitations.

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 measures:

- The number of IRQ calls added per code rooted in a single 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 number of main loops which CBMC unwinds in the `SchedulerBasicP_Scheduler_taskLoop` method. 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 automatized 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 3.3 came up negative, when allowed two task loops and one IRQ per loop (in the case of *Blink* and *Sense*) and up to eight loop and eight IRQs per loop for *TestDissemination*. We then artificially triggered some positive runs in *TestDissemination*. First, 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 `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. Second, as a means to make visible such incorrect parameter passing even when sanity checks are in place, we cause a large `objectSize` being passed to the `send` call:

```
static void
  DisseminationEngineImplP_sendObject(uint16_t key)
{
  void *object;
  uint8_t objectSize = 0;

  // we cause this to set objectSize to a large value
  object = DisseminationEngineImplP_DisseminationCache_requestData
             (key, &objectSize);

  // which is then sent to a send call
  DisseminationEngineImplP_AMSend_send(
    AM_BROADCAST_ADDR,
    &DisseminationEngineImplP_m_buf,
    sizeof(dissemination_message_t) + objectSize);
}
```

We then introduce an `assert(0)` to strengthen the sanity check in the implementation of `send`:

```
static error_t CC2420ActiveMessageP_AMSend_send([..] uint8_t len)
{
  if(len > CC2420ActiveMessageP_Packet_maxPayloadLength())
  {
    assert(0);
    return ESIZE;
  }
}
```

Whenever `ESIZE` would be returned, the assertion is now also reachable and reports failure, and a program trace to track the location of the faulty call.

Constant-address dereference

A potential, secondary source of errors in embedded software is that of dereferencing constant memory addresses. While null pointers can still be erroneous (as exemplified by Section 3.6), dereferencing constant, low pointers is generally expected from embedded code. To enforce a degree of *safety* when dereferencing constants is envolved, 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 [11], which has programmers explicitly mark legal dereferences of constants with a *trusted type*, and thus make null-pointer dereferences visible.

Verification times

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*), and generate its boolean formula and have this verified by the SAT solver (i.e., the *decision procedure runtime*).

Fig. 3.3 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 generated regarding code in function `SchedulerBasicP_pushTask`. 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 task loop; the first run unwinds the task loop once, and the second twice.

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 3.5, one higher than the number of loop iterations, e.g., written as a list of `loop number:loop bound`:

```
--unwindset 0:9,1:2,2:2,3:2,4:9,6:1,7:17,8:17,9:9,11:4,12:1 [..]
```

To give a view 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 from the C program with annotated assertions. Table 3.4 gives a fragment of such a verification condition for the function where the above claim resides; left is the original C code for the function; right is the relevant succession of assignments of bit vectors in CBMC's program translation into verification conditions. Both listings end with giving the claim, in C and boolean form, respectively.

When solving the overall program formula after bit blasting, the conjunctive normal form writes as 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-task-loop verification runs took up to 55 minutes.

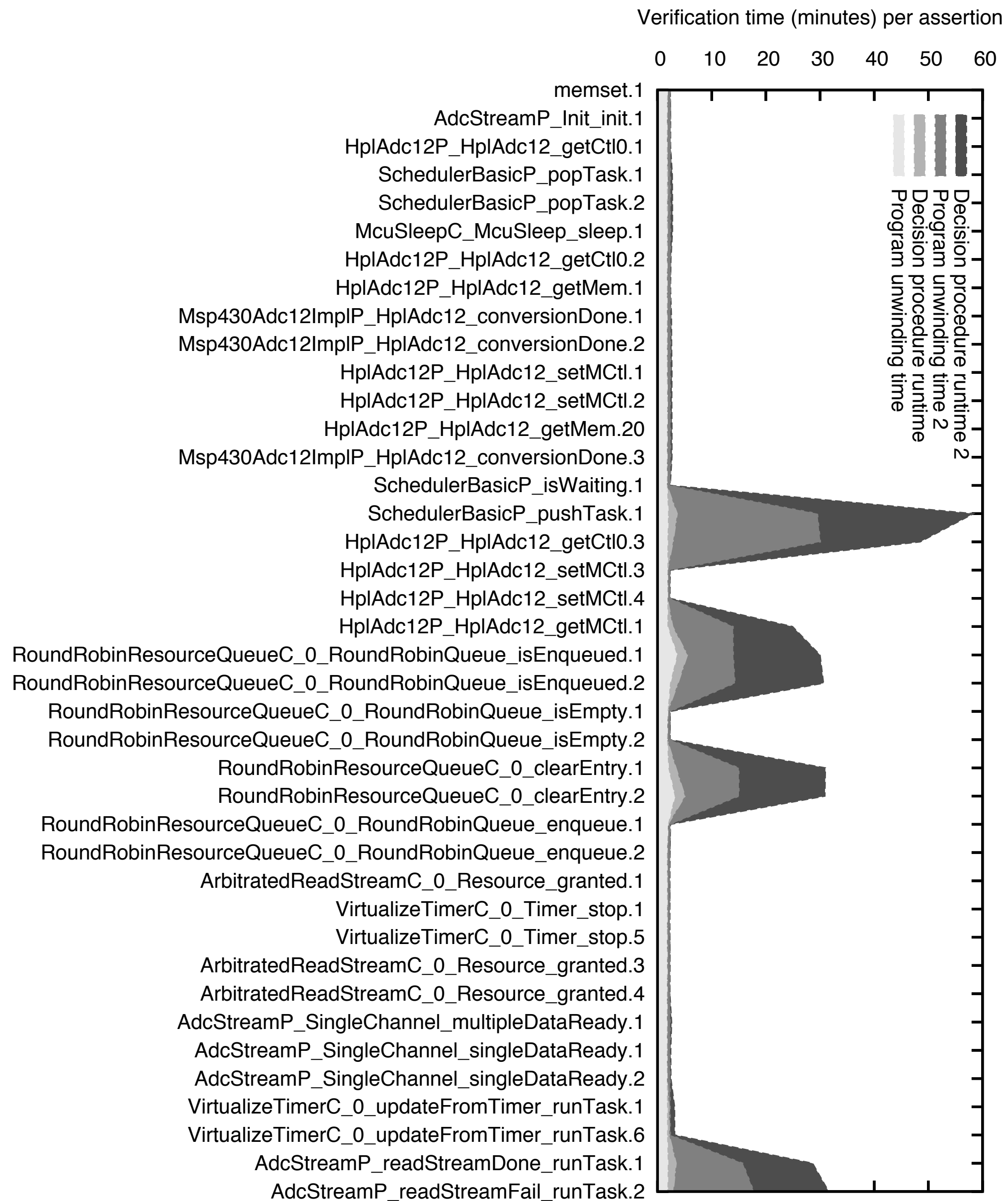


Figure 3.3: Verification times for selected memory-violation assertions in *Sense*

Table 3.4: Fragment of verification condition for *Sense*

```
__inline static uint8_t SchedulerBasicP_popTask( void )
{
    if(SchedulerBasicP_m_head != SchedulerBasicP_NO_TASK)
    {
        uint8_t id = SchedulerBasicP_m_head;
        SchedulerBasicP_m_head =
            SchedulerBasicP_m_next[SchedulerBasicP_m_head];
        if(SchedulerBasicP_m_head == SchedulerBasicP_NO_TASK)
        {
            SchedulerBasicP_m_tail = SchedulerBasicP_NO_TASK;
        }
        SchedulerBasicP_m_next[id] = SchedulerBasicP_NO_TASK;
        return id;
    }
    else
        return SchedulerBasicP_NO_TASK;
}

--
Claim SchedulerBasicP_popTask.1:
  line 5859 function SchedulerBasicP_popTask
  array 'SchedulerBasicP_m_next' upper bound
  (unsigned int)SchedulerBasicP_m_head < 8
```

```
\guard#6 == !(SchedulerBasicP_m_head#8 != 255)
id@2#1 == SchedulerBasicP_m_head#8
SchedulerBasicP_m_head#9 ==
    SchedulerBasicP_m_next#12[SchedulerBasicP_m_head#8]
\guard#7 == !(SchedulerBasicP_m_head#9 == 255)
SchedulerBasicP_m_tail#10 == 255
SchedulerBasicP_m_tail#11 ==
    (\guard#7 ? SchedulerBasicP_m_tail#9 : SchedulerBasicP_m_tail#10)
SchedulerBasicP_m_next#13 ==
    (SchedulerBasicP_m_next#12 WITH [id@2#1:=255])
return_value_SchedulerBasicP_popTask$2@2#1 == id@2#1
return_value_SchedulerBasicP_popTask$2@2#2 ==
    return_value_SchedulerBasicP_popTask$2@2#0
return_value_SchedulerBasicP_popTask$2@2#3 == 255
return_value_SchedulerBasicP_popTask$2@2#4 ==
    (!\guard#6 ? return_value_SchedulerBasicP_popTask$2@2#1
    : return_value_SchedulerBasicP_popTask$2@2#3)

--
{1} !\guard#4 && !\guard#8 && !\guard#10 =>
    (unsigned int)SchedulerBasicP_m_head#11 < 8
```

Chapter 4

Related Work

4.1 Runtime safety

Most existing solutions against software errors in sensor operating systems act at *run-time*, 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 one or another diagnosis or recovery measure is taken (which usually consists in reporting the error and rebooting, as summarized in Table 4.1). While a necessary solution to ensure safe execution in all execution contexts, runtime error detection for deployments is necessarily followed by the expensive redeployment of software, and could instead be preceded by offline means of error detection to save on redeployment efforts.

Table 4.1: Runtime diagnosis and recovery solutions to software errors

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

Safe TinyOS [11] detects memory and type violations in deployed, running 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 doesn't 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 envolves 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 by Deputy into a program instrumented with calls to a failure routine `if(i >= len) deputy_fail()`. Statements accessing a fixed address such as `0x0031` (classic memory violations, except for the case of embedded systems) 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 [6] 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* [1] write specification-like type annotations which define the "correct" use of TinyOS 1.x nesC component 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 [30]; 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 [26] 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 appli-cation-specific assertions. The checks for deadlock and livelock involve a manually added timer to check that each thread goes through its duty cycle.

4.2 Hybrid approaches: 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 [38], MSPsim [15], and WSim [16]. 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 4.2; as it simulates hardware, the technique is mostly OS-independent.

Table 4.2: Sensor platform emulators

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

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 need to act at deployment-time.

4.3 Verification and simulation

A weak case 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 [28], which accurately (but not cycle-accurately) simulates TinyOS applications from their implementation.

TinyOS's own 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 [19] then gave an algorithm for the elimination of such false positives.

SMT solvers are used as backends to ANSI-C model checkers, instead of SAT solvers in [12, 13, 14], 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 mounded model checkers applicable to checking realistic C, [13] 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. [14] adds a state-space-reducing technique for the same verification method; this looks at the modifications suffered 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, [12] 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 [5] described in Chapter 3, 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 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 Since our own contributions, other verification or high-coverage validation methods tackled the more difficult problem of static debugging for sensor network protocols.

KleeNet [33] 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 [33]: 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 distrbuted 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 OS's TCP/IP stack, one of which deadlocked a network node.

On a similar note to KleeNet, T-Check [29] uses *random walks* and execution-driven, depth-bounded *explicit-state model checking*. It builds on the TOSSIM [28] simulator for TinyOS, and inherits its emulation of hardware at the TinyOS interface, rather than at the hardware register level; this precludes 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'. 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 [32] is a model-checking based verification tool for Contiki applications; as a plus over our `tos2cprover`-based tool [5], 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 [5], or remodelling network communication (similar to our [4]). LTL specifications are inputted to the Bogor model checker, and Anquiro is able to find a network configuration in which a dissemination protocol doesn't reach all nodes.

Other static approaches *Insense* [34] 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 [4] described in Chapter 2) as Insense components and matching channels. For the purpose of verification, the components and channels are translated in Promela and model checked against LTL properties by SPIN [21]; 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 [23] 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 thoroughly applied over the demo applications available with TinyOS.

Bibliography

[1] Will Archer, Philip Levis, and John Regehr. Interface contracts for TinyOS. In *Proceedings of the international conference on Information Processing in Sensor Networks (IPSN)*, pages 158–165. ACM, 2007.

[2] Atmel. Atmel 8051 microcontrollers hardware manual. `www.atmel.com,doc4316.pdf`, 2008.

[3] Atmel. 8-bit AVR microcontroller with 128K bytes in-system programmable Flash. `www.atmel.com,doc2467.pdf`, 2009.

[4] Doina Bucur and Marta Kwiatkowska. Bug-free sensors: The automatic verification of context-aware TinyOS applications. In *Proceedings of the European conference on Ambient Intelligence (AmI)*, volume LNCS 5859, pages 101–105. Springer Verlag, 2009.

[5] Doina Bucur and Marta Z. Kwiatkowska. Software verification for tinyos. In *IPSN*, pages 400–401, 2010.

[6] Yang Chen, Omprakash Gnawali, Maria Kazandjieva, Philip Levis, and John Regehr. Surviving sensor network software faults. In *Proceedings of the Symposium on Operating System Principles (SOSP)*. ACM, 2009.

[7] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[8] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based Predicate Abstraction for ANSI-C. In *TACAS*, volume 3440 of *LNCS*, pages 570–574. Springer Verlag, 2005.

[9] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[10] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[11] Nathan Cooprider, Will Archer, Eric Eide, David Gay, and John Regehr. Efficient memory safety for TinyOS. In *Proceedings of the conference on Embedded Networked Sensor Systems (SenSys)*, pages 205–218. ACM, 2007.

[12] Lucas Cordeiro, Bernd Fischer, Huan Chen, and Joao Marques-Silva. Semiformal verification of embedded software in medical devices considering stringent hardware constraints. In *Proceedings of the International Conference on Embedded Software and Systems*, 2009.

[13] Lucas Cordeiro, Bernd Fischer, and João Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 137–148, 2009.

[14] Lucas Cordeiro, Bernd Fischer, and João Marques-Silva. 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)*, pages 160–169, 2010.

[15] Joakim Eriksson, Adam Dunkels, Niclas Finne, Fredrik Österlind, and Thiemo Voigt. MSPsim – an extensible simulator for MSP430-equipped sensor boards. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, 2007.

[16] Antoine Fraboulet, Guillaume Chelius, and Eric Fleury. 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)*, pages 176–185. ACM, 2007.

[17] David Gay, Phil Levis, and David Culler. Software design patterns for TinyOS. In *Proceedings of the ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES)*, pages 40–49. ACM, 2005.

[18] David Gay, Philip Levis, and Robert von Behren. The nesC language: A holistic approach to networked embedded systems. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 1–11. ACM, 2003.

[19] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *Proceedings of the conference on Programming Language Design and Implementation (PLDI)*, pages 1–13. ACM Press, 2004.

[20] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *SIGPLAN Not.*, 35(11):93–104, 2000.

[21] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003.

[22] Kevin Klues, Chieh-Jan Liang, Jeongyeup Paek, Răzvan Musăloiu, Ramesh Govindan, Andreas Terzis, and Philip Levis. TOSThreads: Safe and Non-Invasive Preemption in TinyOS. In *Proceedings of the ACM conference on Embedded Networked Sensor Systems (SenSys)*. ACM, 2009.

[23] Nupur Kothari, Todd Millstein, and Ramesh Govindan. Deriving state machines from TinyOS programs using symbolic execution. In *Proceedings of the international conference on Information Processing in Sensor Networks (IPSN)*, pages 271–282. IEEE, 2008.

[24] Daniel Kroening. Formal verification. `http://www.cprover.org/`.

[25] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.

[26] Veljko Krunic, Eric Trumpler, and Richard Han. NodeMD: Diagnosing node-level faults in remote wireless sensor systems. In *Proceedings of the international conference on Mobile Systems, Applications and Services (MobiSys)*, pages 43–56. ACM, 2007.

[27] Philip Levis, David Gay, Vlado Handziski, Jan-Hinrich Hauer, Ben Greenstein, Martin Turon, Jonathan Hui, Kevin Klues, Cory Sharp, Robert Szewczyk, Joe Polastre, Philip Buonadonna, Lama Nachman, Gilman Tolle, David Culler, and Adam Wolisz. T2: A second generation OS for embedded sensor networks. Technical Report TKN-05-007, Technische Universität Berlin, 2005.

[28] Philip Levis, Nelson Lee, Matt Welsh, and David E. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proceedings of the ACM conference on Embedded Networked Sensor Systems (SenSys)*, pages 126–137, 2003.

[29] Peng Li and John Regehr. T-Check: bug finding for sensor networks. In *Proceedings of the 9th International Conference on Information Processing in Sensor Networks (IPSN)*, pages 174–185. ACM, 2010.

[30] Volker Menrad, Miguel Garcia, and Sibylle Schupp. Improving TinyOS developer productivity with statecharts. In *Proceedings of the Workshop on Self-Organising Wireless Sensor and Communication Networks*, 2009.

[31] 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.

[32] Luca Mottola, Thiemo Voigt, Fredrik Österlind, Joakim Eriksson, Luciano Baresi, and Carlo Ghezzi. Anquiro: Enabling efficient static verification of sensor network software. In *Proceedings of Workshop on Software Engineering for Sensor Network Applications (SESENA) ICSE(2)*, 2010.

[33] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. KleeNet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of the 9th International Conference on Information Processing in Sensor Networks (IPSN)*, pages 186–196, 2010.

[34] Oliver Sharma, Jonathan Lewis, Alice Miller, Al Dearle, Dharini Balasubramaniam, Ron Morrison, and Joe Sventek. Towards verifying correctness of wireless sensor network applications using Insense and SPIN. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*, pages 223–240, Berlin, Heidelberg, 2009. Springer-Verlag.

[35] N. Sörensson and N. Eén. MiniSat — a SAT solver with conflict-clause minimization. In *Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 3569 of *LNCS*. Springer-Verlag, 2005.

[36] Texas Instruments. MSP430x1xx family — user's guide (Rev. F). `www.ti.com, slau049f.pdf`, 2006.

[37] The Contiki Operating System. `http://www.sics.se/contiki/`; accessed 2010.

[38] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Proceedings of the 4th international symposium on Information processing in sensor networks (IPSN), Demo session*, pages 67–72. IEEE Press, 2005.

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

[40] Upkar Varshney. Pervasive healthcare and wireless health monitoring. *Mobile Networks and Applications (MONET)*, 12(2-3):113–127, 2007.