Static Analysis of Device Drivers in TinyOS

Abdelraouf Ouadjaout, Noureddine Lasla, Miloud Bagaa and Nadjib Badache
CERIST Research Center, Algiers, Algeria
{aouadjaout, nlasla, bagaa, badache}@mail.cerist.dz

***Abstract*—In this paper, we present SADA, a static analysis tool to verify device drivers for TinyOS applications. Its broad goal is to certify that the execution paths of the application complies with a given hardware specification. SADA can handle a broad spectrum of hardware specifications, ranging from simple assertions about the values of configuration registers, to complex behaviors of possibly several connected hardware components. The hardware specification is expressed in BIP, a language for describing easily complex interacting discrete components. The analysis of the joint behavior of the application and the hardware specification is then performed using the theory of Abstract Interpretation. We have done a set of experiments on some TinyOS applications. Encouraging results are obtained that confirm the effectiveness of our approach.**

I. Introduction

Robust device drivers play an important role in the reliability of operating systems. They are designed to encapsulate the low-level mechanisms needed to access the device's functions, in order to present a more abstract programming interface for other software components. The implementation of such mechanisms is generally a laborious and error-prone task. This difficulty raises from the fact that programmers must follow very specific *software/hardware interaction patterns* as described in the device datasheet, that involve bit-wise manipulations of MCU registers and often require handling asynchronous hardware answers via interrupts.

In this paper, we present our static analysis tool SADA (*Static Analyzer with Device Abstraction*) for verifying the correctness of device drivers in TinyOS applications. The basic idea is to (*i*) give programmers a means to easily express how the software should interact with the hardware, and then (*ii*) statically check that the different execution paths conforms with such specifications. For example, the correct CC2420 boot sequence specifies that "*the program should never turn on the oscillator before starting the voltage regulator*". These different actions, namely starting the oscillator and the voltage regulator, are performed via specific CC2420 configuration registers, which are accessed through an SPI bus managed by the MCU. Therefore, the program should ensure a reliable SPI communication through which correct configuration commands are performed. SADA can track all these low-level interactions and verify if the program behaviors perform the correct patterns of actions. Existing verification tools for TinyOS, such as [1], can not handle complex specifications about desired software/hardware interactions. This limitation is due to the fact that they analyze the program in *isolation* without considering the state and the behavior of the hardware, which may change in reaction to software/hardware interactions.

To facilitate the description of the desired software/hardware interactions as well as the behavior of the hardware itself, we use BIP [2], a language for expressing complex interacting systems in a compositional manner. Basically, a BIP model represents a set of *Communicating Sequential Processes*. The behavior of a sequential process is defined by an *atom*, which is a labeled transition system extended with data. Each atom can export an external interface defining which synchronization events (called *ports* in BIP) the atom can expect. When linking ports of distinct atoms, we define synchronization rendez-vous, called *interactions*, which represent synchronized transitions among several atoms with eventual data transfer.

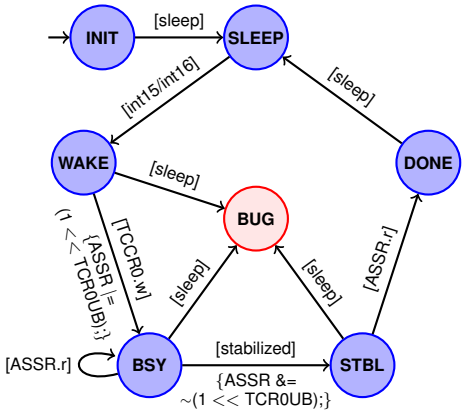

Fig. 1. A BIP atom describing Timer0 stabilization in ATmega128. $[E]$ denotes a synchronization event and $\{A\}$ a transition action. The special event stabilized expresses the asynchronous termination of the stabilization phase.

Fig. 1 depicts an example of a BIP model describing a hardware specification as used in SADA. The model expresses a stabilization condition of Timer0 in an ATmega128L MCU. Basically, when this chip is configured in asynchronous mode with an external 32Khz crystal, one should ensure that the time between entering a timer interrupt (number 15 or 16) and returning to sleep mode must be greater than one crystal cycle. To guarantee this, the MCU datasheet recommends to write to the control register `TCCR0`, for example, and wait until the update flag `TCR0UB` in the status register `ASSR` returns to zero. This *execution pattern requirement* can be automatically verified at *compile-time* by SADA against any TinyOS application to guarantee that no execution path leads to the BUG state.

To verify that no error state in a hardware model is reachable, we use the theory of Abstract Interpretation [3] which is a successful formal method for building sound and computable over-approximations of the semantics of discrete dynamic systems. SADA performs a forward reachability analysis that collects the reachable states of the joint dynamics of the program and the hardware model. We have implemented several

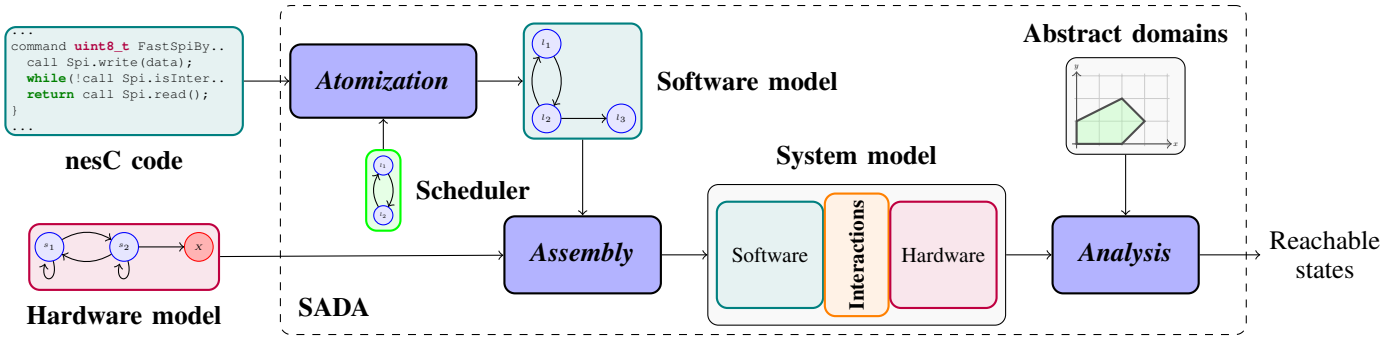

Fig. 2. Analysis steps in SADA: we *atomize* the nesC to obtain a BIP atom, we *assemble* it with the hardware model and the resulting product is then *analyzed*.

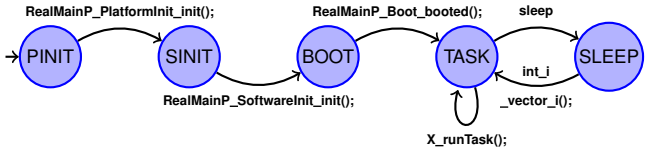

Fig. 3. A simple scheduler for a typical TinyOS application.

TABLE I. RESULTS

	LOC	Time			Memory
		Inlining	Equations	Analysis	
Null	1118	0.004s	0.004s	0.06s	17MB
Blink	2376	0.11s	0.4s	8.5s	102MB
RadioCountToLeds	12793	115.0s	21.8s	272.06s	668MB

abstract domains for manipulating complex C constructs, such as arbitrary pointer arithmetic and bit-wise operation, which extend the previous works of Miné [4].

II. Tool Architecture

We illustrate in Fig. 2 the global architecture of SADA. The analysis is performed in three steps: *atomization*, *assembly* and *analysis*. The first two steps aim at unifying the description of the whole system (software + hardware) as a single BIP model. The third step statically analyzes this global model to obtain an over-approximation of the reachable states.

Atomization starts by invoking the `ncc` compiler in order to transform the nesC code into a C code which is then inlined. To transform the resulting C code into an atom, we have introduced the notion of a *scheduler*. This special atom defines the pattern of all program paths to be analyzed. In other words, it describes how the different program units (*i.e.* initialization functions, tasks and interrupt vectors) are orchestrated during execution. Fig. 3 shows a simple scheduler for a TinyOS application. This scheduler formalizes the fact that an application first initializes the platform before booting the application. Afterwards, waiting tasks are consumed before entering to sleep. When an interrupt arrives, the corresponding vector is executed and generated tasks are consumed again. The final software atom is obtained by replacing the calls to the different program units in the scheduler (such as `RealMainP__Boot__booted()` or `__vector_11()`) with their implementation in the previously obtained C code.

The second step consists in assembling the software and hardware models to obtain a system model describing how these two sub-systems evolve and interact. To do so, we need to locate and extract all low-level interactions occurring in these sub-systems: read/write operations to MCU registers, sleep transitions and interrupts firing. We connect interactions that are present in both sub-systems and soundly ignore the other ones. For example, whenever the program performs a read to a register that is not handled by the hardware model, we replace this read with a top value during the analysis.

The last step consists in analyzing statically the dynamic behavior of the system using Abstract Interpretation. We build the semantic equations and compute a fixpoint solution, which represents an over-approximation of the reachable states. If no BUG location is reachable, SADA deduces that the property is preserved. However, nothing can be said if BUG is reachable due to the undecidability of the verification problem.

III. Experiments

We have implemented SADA using the OCaml language. Experiments were performed on a 2.66GHz Intel Core2 Duo CPU with 2GB memory running Linux 2.6.32. We have used SADA to verify the timer stabilization condition against three applications available in the TinyOS distribution: Null, Blink and RadioCountToLeds. TABLE I gives the obtained results. In all cases, SADA was able to certify the correctness of the applications w.r.t. the specified condition. We have also injected some bogus manipulations inside the stabilization procedure of the Timer0 driver, such as polling on a wrong bit in `ASSR`. Our tool succeeded to catch the bugs.

Acknowledgments

We are deeply indebted to Prof. Saddek Bensalem for allowing us to discover BIP during several visits to Verimag.

References

[1] D. Bucur and M. Z. Kwiatkowska, "Software verification for tinyos," in *IPSN'10*, 2010, pp. 400–401.

[2] A. Basu, M. Bozga, and J. Sifakis, "Modeling heterogeneous real-time components in bip," in *SEFM'06*, 2006, pp. 3–12.

[3] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL'77*, 1977, pp. 238–252.

[4] A. Miné, "Abstract domains for bit-level machine integer and floating-point operations," in *WING'12*, 2012, p. 16.