The Journal of Systems and Software 120 (2016) 114–132

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Static analysis by abstract interpretation of functional properties of device drivers in TinyOS

Abdelraouf Ouadjaout[a,b,c,d,*], Antoine Miné[c,d], Noureddine Lasla[a,b], Nadjib Badache[a,b]

[a] CERIST Research Center, Algiers, Algeria
[b] USTHB University, Algiers, Algeria
[c] École Normale Supérieure, Paris, France
[d] University Pierre and Marie Curie, LIP6, Paris, France

ARTICLE INFO

Article history:
Received 24 May 2015
Revised 6 June 2016
Accepted 21 July 2016
Available online 27 July 2016

Keywords:
Static analysis
abstract interpretation
wireless sensor networks
device drivers

ABSTRACT

In this paper, we present a static analysis by Abstract Interpretation of device drivers developed in the TinyOS operating system, which is considered as the *de facto* system in wireless sensor networks. We focus on verifying user-defined functional properties describing safety rules that programs should obey in order to interact correctly with the hardware. Our analysis is sound by construction and can prove that all possible execution paths follow the correct interaction patterns specified by the functional property. The soundness of the analysis is justified with respect to a preemptive execution model where interrupts can occur during execution depending on the configuration of specific hardware registers. The proposed solution performs a modular analysis that analyzes every interrupt independently and aggregates their results to over-approximate the effect of preemption. By doing so, we avoid reanalyzing interrupts in every context where they are enabled which improves considerably the scalability of the solution. A number of partitioning techniques are also presented in order to track precisely some crucial information, such as the hardware state and the tasks queue. We have performed several experiments on real-world TinyOS device drivers of the ATmega128 MCU and promising results demonstrate the effectiveness of our analysis.

1. Introduction

Wireless sensor networks are autonomous systems composed of a set of tiny embedded nodes with limited computational power that can communicate with each other using short range wireless transmissions. Using distributed routing algorithms, these systems are able to establish a multihop network in order to cover large geographic areas. The main aim of this technology is to remotely monitor (possibly harsh) environments by equipping nodes with specific sensors and propagating their measurements through the *ad hoc* network towards the end-users. Wireless sensor networks have gained great popularity due to their wide variety of applications (such as habitat and health monitoring, smart cities, *etc*) and are considered as a key enabler of the future Internet of Things (Atzori et al., 2010).

The correct operation of these systems relies on the robustness of the programs controlling the nodes. These programs are composed of a hierarchy of software components with different roles as depicted in Fig. 1. As we can see from this architecture, device drivers play a central role among the other components. For instance, the kernel relies on device drivers in order to manage the power of the MCU (*Microcontroller Unit*) and configure the interrupt masks. The networking protocols interact heavily with the device drivers in order to exchange packets with other nodes through the wireless transceiver and retrieve the signal quality of communication links. Finally, device drivers offer to user applications the access to sensor readings in addition to other hardware components such as EEPROM chips for external data storage.

Therefore, it is vital to verify the reliability of device drivers since a single software error may affect the operation of the entire network as all the sensors run the same software. We can divide these failures into two categories depending on the semantic layer of the error. On the one hand, the driver can crash due to a *generic language error* by violating the specifications of the programming language, such as out-of-bounds array access and null pointer dereferences. This type of errors has been tackled by most existing

* Corresponding author.
E-mail addresses: aouadjaout@cerist.dz, ouadjaout@gmail.com (A. Ouadjaout), antoine.mine@lip6.fr (A. Miné), nlasla@cerist.dz (N. Lasla), badache@cerist.dz (N. Badache).

http://dx.doi.org/10.1016/j.jss.2016.07.030
0164-1212/

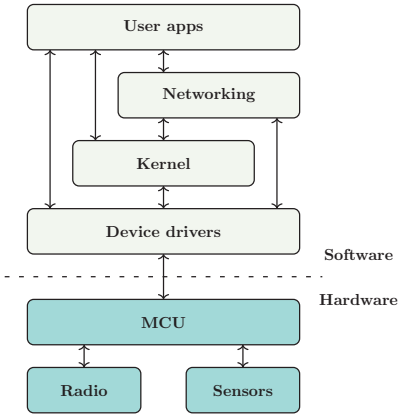

Fig. 1. Simplified software architecture of a typical sensor program.

driver verification solutions (such as Regehr (2005); Brauer et al. (2010); Bucur and Kwiatkowska (2011); Kroening et al. (2015)). On the other hand, *logic errors* are related to the way the driver and its device interact. They occur when this communication transgresses the manufacturer's rules that specify how to correctly access the hardware functionalities. Existing tools offer developers the possibility to instrument their source code with assertions in order to track the proper evolution of their driver. However, these assertions should be inserted manually and may require modifications if the program is changed. In addition, assertions about program variables and hardware registers may not be appropriate to easily express some requirements such as complex temporal properties (*i.e.*, an ordering of actions to perform).

Additionally, a major challenge hampering the verification of device drivers is *concurrency* that induces generally a dramatically large space of possible execution paths that computers can not represent nor manipulate. We can find two distinct forms of concurrency in wireless sensor systems. *Interrupts* are the main source of concurrency and can lead to complex execution traces and unexpected situations not considered during design time since they can preempt the execution of the program at any moment. The second concurrency form is related to hardware operations that can be performed in parallel to the execution of the program. For example, the MCU contains several sub-systems that can answer the program's requests in an asynchronous way without suspending its execution. Generally, the hardware manufacturer provides specific guidelines for driver developers to track the concurrent evolution of the hardware state. Existing verification tools consider only the first form of concurrency and are therefore inadequate to analyze effectively the behavior of the driver in the presence of these asynchronous hardware operations.

In this paper, we propose a static analysis for verifying the absence of logic errors in device drivers by considering all possible execution paths emerging from both forms of concurrency. Our analysis is tailored for programs running the TinyOS operating system (Levis et al., 2004), which is the most popular system for this technology. The analysis is performed *statically*, which means that it is executed at compile time in order to ensure that the program is correct before deploying it. In order to find the logic errors, we require the developer to provide a *functional property* – expressed as a special type of register automata (Kaminski and Francez, 1994) – that specifies the pattern of correct hardware interactions for performing a particular action, along with forbidden hardware states that should be avoided. The property is tied to the hardware specification, not to the driver, hence it can be reused without modification to analyze different versions of a driver, or even completely different implementations of it, which we illustrate in our experimental results. In this work, we exemplify the applicability of our approach on several drivers of the ATmega128 MCU found in many popular sensor platforms such as MicaZ and Waspmote. Nevertheless, the analysis is not restricted to this hardware platform and can be easily extended to other low-power architectures, such as MSP430 or ARM Cortex M0.

The analysis is developed within the theory of Abstract Interpretation (Cousot and Cousot, 1977), a general and successful formal framework for constructing sound approximations of undecidable (or too costly) problems about the semantics of large programs (Blanchet et al. (2002); Cousot et al. (2005)). Our analysis computes a conservative over-approximation of the reachable states of the system (including program variable values and hardware state) for all possible executions. No behavior, in particular, no hardware error is omitted, which makes our analysis sound by construction and able to certify the correctness of the driver w.r.t. to the specification. Our approach can suffer however from false alarms due to the over-approximations necessary to scale up. Note that other state-of-the-art formal analyzers of interrupt-based programs are generally based on bounded model checking techniques that are less vulnerable to the problem of false alarms, but can not provide guarantee about entire search space coverage and thus can suffer from "false negative" (*i.e.*, missing actual bugs), which makes them more adequate to bug finding than certification. That being said, in practice, our analysis can achieve a high precision level thanks to carefully constructing designed abstractions adequate to driver verification and TinyOS semantics.

The remaining of the paper is organized as follows. Section 2 provides a description of TinyOS and how the different software components are orchestrated during execution. An example of a TinyOS driver is discussed in Section 3, where we show also how we express a hardware functional property related to this driver. Section 4 provides a short introduction to the theory of Abstract Interpretation. The details of our analysis are provided in Sections 6 and 7. To simplify the presentation of our abstract interpreter, we proceed in two steps. First, we present in Section 6 a restricted version of our analysis limited to sequential executions where interrupt preemption is not supported. This simplification will allow us to focus on the needed abstraction techniques for dealing with the hardware state and TinyOS scheduler. After that, we extend this techniques in Section 7 in order to handle arbitrary interrupts preemption during execution. Experimental results of the analysis of real-world drivers are presented in Section 8. We discuss in Section 9 the related work and we end the paper in Section 10 by a conclusion.

2. TinyOS

TinyOS is an event-based operating system developed by Levis et al. (2004) for low-power wireless sensor nodes. Thanks to its small memory footprint, TinyOS can run on tiny constrained MCUs that have 2–10 KB of SRAM and 32–128 KB of flash memory. It supports a variety of hardware platforms with built-in device drivers, networking protocols, security mechanisms, *etc*. TinyOS programs are written in the nesC language (Gay et al., 2003), a dialect of C that offers a modular programming paradigm for flexible organization of software components. During compilation, nesC programs are translated into equivalent C programs using the `ncc` compiler.

TinyOS programs are driven by a two-level preemption system with the concepts of interrupts and tasks. *Interrupts* represent the high priority preemption level. They play an important role in designing power-efficient programs and are used to free up the MCU from actively waiting for the occurrence of a particular event. During these waiting periods, the microcontroller can either enter various sleep modes to save energy or execute other *waiting*