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*

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.

Keywords: Static analysis, abstract interpretation, wireless sensor networks, device drivers.

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 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

*Corresponding author
Email addresses: aouadjaout@cerist.dz (Abdelraouf Ouadjaout), antoine.mine@lip6.fr (Antoine Miné), nlasla@cerist.dz (Noureddine Lasla), badache@cerist.dz (Nadjib Badache)

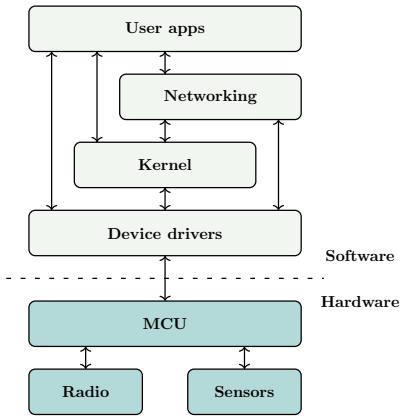


Figure 1: Simplified software architecture of a typical sensor program.

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 functions* to save time. *Tasks* are a special feature of nesC that provides this concept of *waiting functions*. This mechanism allows postponing the execution of a function in order to let other tasks execute. That is, when a task is *posted*, the TinyOS scheduler puts it in a *task queue* and the execution of the current function is resumed. The scheduler, at specific moments, checks its task queue in order to *consume* the posted tasks. Tasks run at low priority and can not preempt each other, while interrupts can preempt the execution of tasks or other interrupts.

To explain further this execution model, we show in Fig. 2 the different steps of a TinyOS program lifecycle. These steps can be divided into two main phases: the initialization phase and the infinite loop. The initialization phase is responsible for bootstrapping the different software components. At the beginning, the kernel and the device drivers are initialized. These steps are executed without enabling interrupts so the system is started in a controlled manner. After that, the TinyOS kernel consumes the tasks that have been posted by device drivers and terminates the initialization phase by starting the user applications. This final step is executed with interrupts being enabled because some drivers rely on interrupts for proper operation.

The second phase is the infinite loop that constitutes the most important proportion of the program's lifetime. This phase begins by consuming the previously posted tasks. When the tasks queue becomes empty, the MCU can enter the sleep mode in order to save energy while waiting for interrupts. After the occurrence of an interrupt, the MCU executes its corresponding handler function. The particular sequence *tasks-sleep-interrupts* forms the body of the infinite loop which is repeated indefinitely until the shutdown of the system. It is important to note that interrupts do not occur only during sleep periods, but they can preempt the execution of the program in every control location when specific conditions on some hardware registers are met, which will be discussed in more details in Section 7.

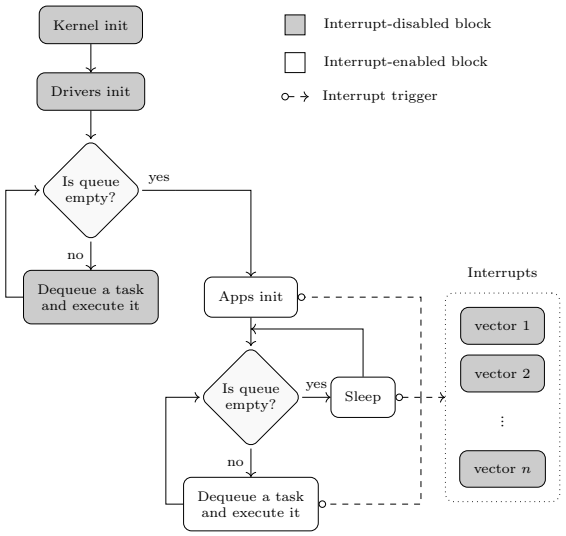


Figure 2: The execution model of a TinyOS program.

3. TinyOS Device Drivers

At the core of the TinyOS operating system, we find a rich library of device drivers for many microcontrollers, transceivers, sensing boards, *etc.* These programs should encapsulate the required sequences of low-level hardware manipulations to activate the requested functionalities. The specifications of these sequences are generally described in data-sheets provided by the manufacturer of the hardware. It is vital to ensure that these *functional properties* are always preserved during runtime.

In this work, we choose to express these properties as a type of register automata, which we call an *Abstract Device Property* (ADP for short), that takes into account the semantics of low-level hardware interactions. An ADP is composed of a finite set of *hardware states* that corresponds to an abstract discretization of the hardware behaviors at specific moments. The dynamics between these states is modeled by a set of transitions that react to the occurrence of special low-level *events*. We can distinguish between four types of events:

Register access events. Given the set of hardware registers $\mathcal{R}$, the events $\{\mathtt{X}^{\diamond} \mid \mathtt{X} \in \mathcal{R}, \diamond \in \{r, w\}\}$ decorate transitions that model the reaction of the device when its registers are accessed by read/write statements issued by the program.

Asynchronous events. Hardware concurrency is an important concept in driver development. Many operations of the MCU sub-systems are performed independently from the program execution flow. A transition decorated with an asynchronous event, that we denote by α, allows us to model the evolution of these concurrent hardware operations.

Interrupt events. Given the set of interrupts $\mathbb{I}$, the events $\{\texttt{int}_i \mid i \in \mathbb{I}\}$ allow the ADP to model the situations where an interrupt can occur. When a transition t is decorated with an event $\texttt{int}_i$, the execution of the interrupt handler and the transition t are performed in a synchronous way.

Sleep event. When the TinyOS kernel terminates the execution of all posted tasks, it configures the MCU to suspend its execution waiting for interrupts. This switching between the active and inactive mode of operation is tracked by the special event `sleep`.

In addition to the occurrence of an event, each transition is decorated with a *guard*, represented as a boolean expression involving hardware registers as variables, that expresses a necessary condition for performing the transition. When both event and guard are satisfied, the ADP can move to the next state after updating the values of its registers using the *action* assignment that labels the transition.

Example 1. Let us take the example of the driver of CC2420, which is a low-power wireless transceiver widely used in sensor motes. It implements the IEEE 802.15.4 standard and can be controlled via a SPI serial bus[1]. The specifications of the ATmega128 (Atmel (2011)) stipulate many rules to establish a correct SPI data exchange. Let us take the example of two major rules:

- No byte can be sent by the master if the bits `MSTR` and `SPE` are not set in the control register `SPCR`.
- To exchange data over the bus, the master must write into the `SPDR` data register. The transfer is handled by the MCU in an asynchronous way and therefore, the master must wait until the termination of the operation. To do so, it should continuously poll the status flag `SPIF` in the `SPSR` status register, which will be cleared by the MCU at the end of the transfer.

Fig. 3 shows the ADP $\mathcal{A}_{\texttt{SPI-TX}}$ for the previous two SPI rules. We symbolize a transition as an arrow decorated with three fields (**e**, g, **a**) representing respectively the event, the guard and the action. Initially, the automaton is put in `OFF` state where data transfer is not allowed. The forbidden data transfer is modeled by a transition to the special state `BUG` decorated with the write event $\texttt{SPDR}^w$ and

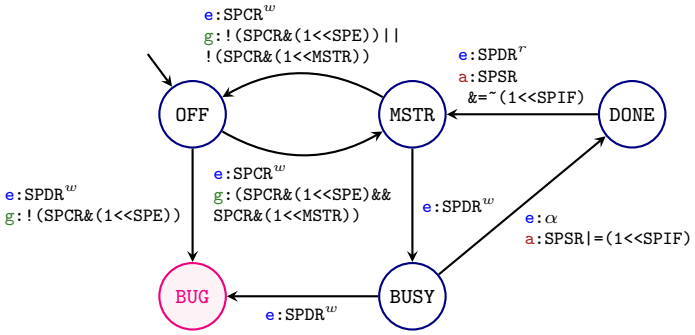


Figure 3: The abstract device property $\mathcal{A}_{\texttt{SPI-TX}}$ modeling a correct SPI transfer for the ATmega128 microcontroller.

guarded by the condition `!(SPCR&(1<<SPE))` which means that the forbidden write operation is performed when the bus is not enabled. When the program modifies the `SPE` and `MSTR` bits in the control register `SPCR`, the automaton enters the `MSTR` state. Putting data in the register `SPDR` starts the SPI communication and the bus becomes `BUSY`. During this state, no other communication can take place. The termination of the transfer is modeled with the asynchronous event α that can occur at any moment during the subsequent program execution. When this event occurs, the flag `SPIF` in the status register `SPSR` is set in order to notify the program. The latter should access the data register `SPDR`, to read the byte sent by the slave for example, which clears the `SPIF` bit and moves the ADP to `MSTR` state. □

In Figures 4(a) and 4(b), we illustrate two driver implementations for our example functional property. The first implementation is relatively straightforward and performs an active polling on the status flag `SPIF` until termination of every byte transfer. The second implementation is more involved and exploits the tasks mechanism in order to let the scheduler execute other tasks while waiting for the end of the transfer of bytes. This driver works as follows: it starts by enabling the SPI sub-system before posting the `tx` task. The latter checks if the number of sent bytes is still less then the number of bytes to send. In this case, the next byte is written into the register `SPDR` and the task `check` is posted. This task verifies the status of the `SPIF` bit: if the bit indicates the end of the transfer, the `tx` task is posted again to send the next byte, otherwise the `check` task posts itself to continue the polling mechanism. When the last byte is sent, the task `end` is posted that turns-off the SPI sub-system. The advantage of such a procedure is that the scheduler takes control of the execution flow when the SPI is busy, which is not the case for the first implementation.

These two illustrative examples demonstrate the fact that a same functional property can be implemented with different manners and complexities. Consequently, it is necessary to analyze the *semantics* of these implementations by considering the dynamic behaviors of the pro-

[1]SPI (*Serial Peripheral Interface*) is a serial protocol for byte exchange between devices on a shared electronic bus.

```
1  void send_spi
2   (char* data, char len) {
3    char i = 0, tmp;
4    SPCR |= (1 << SPE)
5     | (1 << MSTR);
6    while (i < len) {
7       SPDR = data[i];
8       while (!(SPSR & (1<<SPIF)));
9       tmp = SPDR;
10      i++;
11   }
12   SPCR &= ~(1 << SPE);
13   ...
14 }
```

(a)

```
1  void send_spi
2   (char* data, char len) {
3    m_data = data;
4    m_len = len;
5    i = 0;
6    post tx();
7    SPCR |= (1 << SPE)
8     | (1 << MSTR);
9  }

18 task void check() {
19   if (SPSR & (1<<SPIF)) {
20     char tmp = SPDR;
21     i++;
22     post tx();
23   } else
24     post check();
25 }

10 task void tx() {
11   if (i >= m_len) {
12     post end();
13     return;
14   }
15   SPDR = m_data[i];
16   post check();
17 }

26 task end() {
27   SPCR &=
28     ~(1 << SPE);
29   ...
30 }
```

(b)

Figure 4: (a) Simplified driver of an SPI data exchange for the ATmega128 MCU. (b) A more complex implementation involving tasks.

gram, since textual pattern matching would be inefficient to catch all possible implementations. However, the dynamic nature of programs can result in complex behaviors difficult to inspect manually. In fact, these behaviors are not computable in general. In this work, we propose to use the theory of Abstract Interpretation to alleviate this problem. In the next sections, we give a short introduction to this theory followed by our formulation of an abstract interpreter that takes into account, on the one hand, the specificities of TinyOS, and on the other one, the evolution of the ADP in reaction to program statements.

4. Abstract Interpretation

Abstract Interpretation is a theory that formalizes the notion of approximation (Cousot and Cousot (1977)). It proposes a general framework for (*i*) handling computable approximations of (possibly infinite) sets and (*ii*) building efficient operators that describe how these approximations evolve in a dynamic system. Basically, the approximations represent a correspondence between the concrete (real) view of the dynamic behaviors and an abstract one that is more efficient and easier to manipulate by programs. A cornerstone feature of this theory is its *soundness guarantee*: the properties proven over the abstract view are also valid for the corresponding concrete elements. During the last decade, the theory of Abstract Interpretation has been widely adopted and successfully applied for the static analysis of program semantics by many commercial tools, such as AbsInt Astrée and MathWorks Polyspace.

To develop an abstract interpreter, we start by defining the *concrete semantics* which is a precise mathematical description of the executions of a program. The concrete semantics is defined by two notions. First, we need a *concrete semantic domain* $\mathcal{D}$, defined generally as a lattice $\langle \mathcal{D}, \sqsubseteq, \sqcup, \bot \rangle$, that provides a representation of program executions. This representation depends on the class of the properties of interest that we are analyzing. For example, if we are interested in performing a reachability analysis, we need to collect the states of the program that may be reached during execution, so we define the semantic domain as $\langle \wp(\Sigma), \subseteq, \cup, \varnothing \rangle$ where Σ represents the set of states. The second notion is the concrete transfer functions $\mathbf{S}[\![s]\!] \in \mathcal{D} \to \mathcal{D}$ defining the effect of a statement s over our semantic domain.

However, the concrete semantics is not computable in general. Therefore, we need to approximate the elements of $\mathcal{D}$ by an *abstract semantic domain* $\langle \mathcal{D}^\sharp, \sqsubseteq_\sharp, \sqcup_\sharp, \bot_\sharp \rangle$ the elements of which are more compact and provide a summary of the elements of $\mathcal{D}$ by ignoring some of their details. This approximation relationship is formalized through a *concretization function* $\gamma \in \mathcal{D}^\sharp \to \mathcal{D}$.

Example 2. Let us consider a simple example of numerical states $\Sigma = V \to \mathbb{Z}$, where V is the set of variables. One way of abstracting the concrete semantic domain $\wp(\Sigma)$ is to use the domain of *intervals* (Cousot and Cousot (1977)) that keeps track of the upper and lower bound for every variable. This abstract domain is defined as $\mathcal{D}^\sharp = V \to (\mathbb{Z} \cup \{-\infty\} \times \mathbb{Z} \cup \{+\infty\})$ with the concretization function $\gamma(X) = \{\lambda v.\ n \mid X(v) = (a,b) \wedge n \in [a,b]\}$. The domain of intervals is very efficient in terms of memory and computations since it needs to save only two numbers for every variable. However, it is not very precise because it can introduce new values and all the relations between variables are ignored. □

In addition to the abstract domain $\mathcal{D}^\sharp$, we need to define the abstract transfer functions that over-approximate the effect of executing the different program statements. For every possible statement s, we build an abstract transfer function $\mathbf{S}[\![s]\!]^\sharp \in \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ that should preserve the soundness condition: $\forall X \in \mathcal{D}^\sharp : \mathbf{S}[\![s]\!] \circ \gamma(X) \subseteq \gamma \circ \mathbf{S}[\![s]\!]^\sharp(X)$.

Example 3. When using the interval abstract domain in analyzing the statement `x = x + 1`, it is sufficient to increment the boundaries of the variable `x`. This can be

formally stated as:

$$\mathbf{S}[\![\texttt{x = x + 1}]\!]^\sharp X \triangleq \\ \quad \text{let } (a,b) = X(\texttt{x}) \text{ in } X[\texttt{x} \to (a+1, b+1)]$$

□

In general, when defining the abstract transfer functions $\mathbf{S}[\![.]\!]^\sharp$, we need to consider only the atomic statements, i.e.: assignments `x = exp` and tests `?exp`. The remaining compound statements, such as conditionals, are defined by structural induction over the syntax tree of the program. For example, when analyzing the statement `if (c) {s1} else {s2}`, we analyze the true-branch `s1` and the false-branch `s2` independently and we merge the results before continuing with the following statement. By doing so, we can build generic analyzers that are parametrized with abstract domains that define only approximations of the atomic statements.

The case of loops is, however, more complex as it requires handling a possibly unbounded number of iterations. Indeed, the semantics of a loop `while (c) {s}` is to repetitively execute the statement `s` until the condition `c` is not verified, which can be expressed with fixpoint iterations as follows:

$$\mathbf{S}[\![\texttt{while (c) s}]\!]^\sharp X = \\ \quad \text{let } X_\star = \text{lfp } \lambda X'.\ X \sqcup \mathbf{S}[\![\texttt{s}]\!]^\sharp \circ \mathbf{S}[\![\texttt{?c}]\!]^\sharp X' \text{ in} \\ \quad \mathbf{S}[\![\texttt{?!c}]\!]^\sharp X_\star$$

where $\text{lfp } \lambda X.\ F(X)$ represents the least element $X_\star$ that satisfies $X_\star = F(X_\star)$ and can be computed using the Kleene theorem as supremum of the sequence $\{F^n(\bot) \mid n \in \mathbb{N}\}$. In other words, to compute the fixpoint $X_\star$, we iteratively build the sequence $X_{n+1} = X \sqcup \mathbf{S}[\![\texttt{s}]\!]^\sharp \circ \mathbf{S}[\![\texttt{?c}]\!]^\sharp X_n$ until $X_{n+1} = X_n$, where $X_0 = \bot_\sharp$. The obtained limit $X_\star$ corresponds to the well-known notion of *loop invariant*, which represents a property satisfied at every loop iteration. However, performing these fixpoint iterations may not terminate in a finite time, which is the case when the lattice of the abstract domain does not verify the ascending chain condition. The theory of Abstract Interpretation introduces the notion of a *widening operator* $\triangledown \in \mathcal{D}^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp$, which is an acceleration technique to over-approximate fixpoint computations. Intuitively, giving the result of two successive iterations X_i and X_{i+1} of the fixpoint computation, the widening operator $\triangledown(X_i, X_{i+1})$ should be chosen in order to stabilize the convergence in a finite number of steps. Formally stated, the abstract transfer function of a loop with widening acceleration becomes:

$$\mathbf{S}[\![\texttt{while (c) s}]\!]^\sharp(X) = \\ \quad \text{let } X_\star = \text{lfp } \lambda X'.\ X' \triangledown (X \sqcup \mathbf{S}[\![\texttt{s}]\!]^\sharp \circ \mathbf{S}[\![\texttt{?c}]\!]^\sharp X') \text{ in} \\ \quad \mathbf{S}[\![\texttt{?!c}]\!]^\sharp X_\star$$

Example 4. Let us analyze the statement `x = 0; while (x < 10) x = x + 1;` with the domain of intervals. Let us denote by X_i the abstract state at the entry of the `while` loop after i fixpoint iterations. If we analyze the loop for the first two iterations, we find that $X_0 = \langle \texttt{x} \mapsto [0,0] \rangle$ and $X_1 = \langle \texttt{x} \mapsto [0,1] \rangle$.

To accelerate the convergence, we can stabilize the changing upper boundary between X_0 and X_1 using the interval widening operator (Cousot and Cousot (1977)) defined as $\triangledown([a,b],[c,d]) = [(c < a) ? -\infty : a, (d > b) : +\infty : b]$. The principle of this operator is to put unstable bounds to infinity, where they cannot evolve anymore, so that the iteration terminates in a finite number of steps (as there are finitely many bounds to put to infinity). Using this over-approximation, and by applying the loop-termination filter `(x >= 10)`, we can easily infer a post-loop abstract state $\langle \texttt{x} \mapsto [10, +\infty[\rangle$.

Note that the obtained result is correct but not optimal. There exists more elaborated techniques, such as widening with thresholds (Blanchet et al. (2002)) as well as decreasing iterations with narrowing (Cousot and Cousot (1992)), to efficiently infer a more precise post-loop abstract state $\langle \texttt{x} \mapsto [10, 10] \rangle$. □

Analyzers built by Abstract Interpretation are sound by construction. This means that no behavior of the program can be omitted during the analysis. This feature provides a guarantee that the analyzer will produce no false negative, which is essential to prove the absence of errors in programs. However, due to the over-approximations introduced by the abstract semantics and the widening operator, spurious errors may be encountered during the analysis, leading to the detection of false positives. These imprecisions can be eliminated by refining the abstractions in order to embed more relevant details, which will be illustrated by the partitioning techniques presented in this work.

5. Assumptions and Notations

Before presenting our abstract interpretation of TinyOS device drivers, we detail the assumptions of the analysis. We assume that the input program has been preprocessed with the `ncc` compiler so that we manipulate the semantically equivalent C program. We denote by $Stmt_C$ the set of statements of this program. As a particularity of TinyOS programs, these C programs have no dynamic allocations nor function pointers. We assume also that there is no recursive functions and no backward `gotos`. Let $\mathbb{T}$ be the set of tasks and $\mathbb{I}$ the set of interrupts of the input program. The statements of these particular elements are defined by the utility function $body \in (\mathbb{T} \cup \mathbb{I}) \to Stmt_C$. Finally, we denote by $I_{ker}, I_{drv}, I_{app} \in Stmt_C$ the initialization routines for the kernel, device drivers and user applications respectively.

We formalize the ADP as a special register automaton $(\mathcal{S}, s_0, \mathcal{R}, \xi, \mathcal{T})$, where:

- $\mathcal{S}$ is the set of hardware states and s_0 is the initial state.
- $\mathcal{R}$ is the set of hardware registers.

- $\xi \triangleq \{\mathtt{X}^\diamond \mid \mathtt{X} \in \mathcal{R}, \diamond \in \{r, w\}\} \cup \{\mathtt{int}_i \mid i \in \mathbb{I}\} \cup \{\alpha, \mathtt{sleep}\}$ is the set of hardware events described previously in Section 3.

- $\mathcal{T} \subseteq \mathcal{S} \times \xi \times \mathcal{S} \times Stmt_C \times Stmt_C$ is the set of transitions where each transition $\tau = (s, e, s', g, a) \in \mathcal{T}$ moves the ADP from state s to s' whenever the event e occurs and the guard g is verified. When the transition is performed, the assignment statement a modifies the required registers.

Since we are analyzing the joint dynamics of the driver, the kernel and the hardware, the statements that affect the global state of the system are not restricted the C atomic statements. Consequently, we consider an extended set $Stmt \triangleq Stmt_C \cup Stmt_H \cup Stmt_Q$ with the following additional statements:

- The set $Stmt_H \triangleq \{\mathsf{event}\, e \mid e \in \xi\} \cup \{\mathsf{event}^\star\, e \mid e \in \xi\}$ consists of the statements that trigger the ADP transitions. The statement $\mathsf{event}\, e$ fires the event e and performs a single transition of the ADP. The statement $\mathsf{event}^\star\, e$ is similar but instead of making a single transition, it continues by firing all asynchronous post-transitions labeled with the event α.

- The set $Stmt_Q \triangleq \{\mathsf{dequeue}\, t \mid t \in \mathbb{T}\} \cup \{\mathsf{post}\, t \mid t \in \mathbb{T}\} \cup \{\mathsf{notask}\}$ refers to the elementary operations for manipulating the TinyOS tasks queue: removing a task from the head of the queue, posting a task at the end of the queue and testing whether the queue is empty.

6. Sequential Executions Analysis

In this section, we describe the design of a static reachability analysis for TinyOS programs by Abstract Interpretation. We start by presenting the analysis of sequential executions where we limit the trigger of interrupts during only sleep periods. This simplification allows us to ignore the preemption of tasks by interrupts, in order to focus on the dynamics of the program related to the interaction with the ADP and the tasks mechanism. In Section 7, we will extend this analysis to take into consideration the arbitrary occurrence of interrupts during execution.

An early version of the sequential executions analysis was briefly described in Ouadjaout et al. (2014) where we supported only numeric abstractions of program variables. In this section, we develop more elaborated domains providing different abstraction levels for handling the evolution of the hardware state of the TinyOS tasks queue. In addition, we present a more efficient analysis method based on structural induction and inspired by the design of the Astrée static analyzer (Cousot et al. (2009)).

6.1. Concrete Semantics

Our concrete semantic domain $\mathcal{D} \triangleq \wp(\mathcal{E})$ is defined as the set of subsets of concrete environments $\mathcal{E} \triangleq \mathcal{M} \times \mathcal{S} \times \mathcal{Q}$ the elements of which provide a complete characterization of the state of the system at a given program location. An environment $\rho = (m, s, q) \in \mathcal{E}$ is divided into three parts describing respectively the memory, the hardware state and the tasks queue.

The *memory environment* $\mathcal{M}$ maintains the values of the program variables, as well as hardware registers that we consider as numeric variables to facilitate their manipulation in usual C expressions. We employ the cell-based representation proposed by Miné (2006a) to deal with complex C data structures and pointer arithmetics. A cell $c = (v, i, \tau) \in \mathcal{C} \subseteq (V \times \mathbb{N} \times T)$ is a tuple encoding an offset i within a host variable v and having a type τ. The memory environment is defined as $\mathcal{M} \triangleq (\mathcal{C} \to \mathbb{Z}) \times (\mathcal{C} \to (V \times \mathbb{N}))$ in which we distinguish between two types of cells: numeric and pointer cells. The numeric cells are mapped to numeric values which range depends on the type of the cell. The pointer cells are considered as tuples describing the target host variable and the offset, in bytes, since the beginning of the target variable. The effect of C statements on $\mathcal{M}$ is given by a set of transfer functions $\mathbf{S}[\![.]\!]_{\mathcal{M}} \in \wp(\mathcal{M}) \to \wp(\mathcal{M})$. For the case of an atomic assignment statement, $\mathbf{S}[\![\mathtt{x = exp}]\!]_{\mathcal{M}}$ evaluates the left hand side expression in every input memory environment and, for every possible value of the expression, returns a new memory environment where the left-hand side target variable of the assignment has been updated. The test transfer function $\mathbf{S}[\![?\mathtt{exp}]\!]_{\mathcal{M}}$ allows filtering the input memory environments to retain only those where the expression `exp` can be evaluated to true. More details about the formalization of the complete semantics of C statements can be found in the work of Miné (2006a).

The *queue environment* $\mathcal{Q}$ provides information about the contents of the tasks queue. There exists two implementations of the queuing system in TinyOS. The first one employs a FIFO ordering of posted tasks with possible redundant occurrences of the same task. This implementation is the default mechanism used in version 1.x of TinyOS. The second implementation considers also a FIFO queue but with the restriction that the queue can not contain two entries for the same task. That is, when a task is posted again before consuming it, the queue is not modified and the second post is ignored. This behavior was chosen for TinyOS version 2.x. In the sequel of this paper, we will describe our analysis using the first implementation, since it is more general and the second one can be easily derived from it. Nevertheless, we will provide in Section 8 the experimental results when using both implementations in order to give an overview about the impact of those strategies on the analysis.

Formally, we define the tasks queue environment as $\mathcal{Q} \triangleq \bigcup_{i \geq 0} \mathbb{T}^i$, where $\mathbb{T}^0 \triangleq \{\varnothing_{\mathcal{Q}}\}$ represents the singleton set of the empty queue $\varnothing_{\mathcal{Q}}$ and $\mathbb{T}^i \triangleq [0, i-1] \to \mathbb{T}$ is the set of finite task sequences $t_0 \dots t_{i-1}$ of length i. We will employ the ordinary concatenation operator $q \cdot t$ (resp. $t \cdot q$) to denote a queue ending (resp. starting) with the task t. In addition, we introduce an auxiliary function `count` $\in$

$$\mathbf{S}[\![\mathsf{notask}]\!]R \triangleq \{(s,m,q) \in R \mid q = \varnothing_{\mathcal{Q}}\}$$
$$\mathbf{S}[\![\mathsf{post}\, t]\!]R \triangleq \{(m,s,q') \mid \exists (m,s,q) \in R : q' = q.t\}$$
$$\mathbf{S}[\![\mathsf{dequeue}\, t]\!]R \triangleq \{(s,m,q') \mid \exists (s,m,q) \in R : q = t.q'\}$$

(We assume that X is the only register occurring in exp)
$\mathbf{S}[\![\mathtt{X = exp}]\!]R \triangleq$
  (Notify the ADP about the read event)
  let $R_1 = \mathbf{S}[\![\mathsf{event}^\star\, \mathtt{X}^r]\!]R$ in
  (Update the register variable with the assignment statement)
  let $R_2 = \{(m',s,q) \mid (m,s,q) \in R_1 \wedge m' \in \mathbf{S}[\![\mathtt{X = exp}]\!]_{\mathcal{M}}(\{m\})\}$ in
  (Notify the ADP about the write event)
  $\mathbf{S}[\![\mathsf{event}^\star\, \mathtt{X}^w]\!]R_2$

$$\mathbf{S}[\![\mathsf{event}^\star\, e]\!]R \triangleq \mathsf{lfp}\ \lambda X.\ \mathbf{S}[\![\mathsf{event}\, e]\!]R \cup \mathbf{S}[\![\mathsf{event}\, \alpha]\!]X$$
$$\mathbf{S}[\![\mathsf{event}\, e]\!]R \triangleq \{(m',s',q) \mid \exists (m,s,q) \in R, \exists (s,e,s',g,a) \in \mathcal{T} : m' \in \mathbf{S}[\![a]\!]_{\mathcal{M}} \circ \mathbf{S}[\![?g]\!]_{\mathcal{M}}(\{m\})\}$$

Figure 5: Concrete transfer functions of sequential executions.

(The set of initial states)
(The queue is empty and all registers are initialized to 0)
let $R_0 = \{(m, s_0, \varnothing_{\mathcal{Q}}) \mid$
  $m \in \mathsf{fold}\ (\lambda r.\lambda R.\ \mathbf{S}[\![r = 0]\!]_{\mathcal{M}} R)\ \mathcal{M}\ \mathcal{R}\}$ in
(Initialize the kernel and the drivers)
let $R_1 = \mathbf{S}[\![I_{drv}]\!] \circ \mathbf{S}[\![I_{ker}]\!]R_0$ in
(Analyze the posted tasks until emptying the queue)
let $R_2 = \mathsf{lfp}\ \lambda R.\ R_1 \cup \bigcup_{t \in \mathbb{T}} \mathbf{S}[\![body(t)]\!] \circ \mathbf{S}[\![\mathsf{dequeue}\, t]\!]R$ in
let $R_3 = \mathbf{S}[\![\mathsf{notask}]\!]R_2$ in
(Initialize the user applications)
let $R_4 = \mathbf{S}[\![I_{app}]\!]R_3$ in

$\mathsf{lfp}\ \lambda R.$ (
  (Analyze the posted tasks)
  let $R_5 = \mathsf{lfp}\ \lambda R'.\ R \cup \bigcup_{t \in \mathbb{T}} \mathbf{S}[\![body(t)]\!] \circ \mathbf{S}[\![\mathsf{dequeue}\, t]\!]R'$ in
  (Move the MCU to sleep mode when no task is posted)
  let $R_6 = \mathbf{S}[\![\mathsf{event}^\star\, \mathtt{sleep}]\!] \circ \mathbf{S}[\![\mathsf{notask}]\!]R_5$ in
  (Analyze the interrupts)
  $R_4 \cup \bigcup_{i \in \mathbb{I}} \mathbf{S}[\![body(i)]\!] \circ \mathbf{S}[\![\mathsf{event}^\star\, \mathtt{int}_i]\!]R_6$
)

Figure 6: Concrete interpreter for sequential executions.

$\mathcal{Q} \times \mathbb{T} \to \mathbb{N}$ giving the number of occurrences of a task in a queue.

We present in Fig. 5 a summary of the most important transfer functions $\mathbf{S}[\![.]\!] \in \wp(\mathcal{E}) \to \wp(\mathcal{E})$ related to hardware state manipulation and TinyOS tasks. The functions $\mathbf{S}[\![\mathsf{post}\, t]\!]$, $\mathbf{S}[\![\mathsf{notask}]\!]$ and $\mathbf{S}[\![\mathsf{dequeue}\, t]\!]$ formalizing the queuing system are straightforward and just alter the queues of the input environment without modifying memory and hardware state. However, handling the effect of hardware interactions is more complex. We give the example of the function $\mathbf{S}[\![\mathtt{X = exp}]\!]$ – where `X` is a register and the expression `exp` contains a read access to the same register – because it represents a frequent pattern in device drivers. For example, it is used to modify a particular bit in a register without altering the other bits, as depicted in the SPI driver in Fig. 4(a). To handle the eventual hardware state changes, we define the functions $\mathbf{S}[\![\mathsf{event}\, e]\!]$ and $\mathbf{S}[\![\mathsf{event}^\star\, e]\!]$ that compute the possible transitions of the ADP in response to an event $e \in \xi$. Intuitively, the function $\mathbf{S}[\![\mathsf{event}\, e]\!]$ computes the effects of the one-step transitions decorated with event e and having valid guards when evaluated in the input environments. The function $\mathbf{S}[\![\mathsf{event}^\star\, e]\!]$ computes the same transitions provided by $\mathbf{S}[\![\mathsf{event}\, e]\!]$ in addition to the subsequent asynchronous transitions decorated with the asynchronous event α. Since the hardware can perform several asynchronous transitions in response to the event e, we need to collect all possible sequences of intermediate states (of arbitrary length) that the hardware can go through during this period. This is the reason for the fixpoint formulation of $\mathbf{S}[\![\mathsf{event}^\star\, e]\!]$, which is similar to the traditional definition of a transitive closure.

Using these transfer functions, we provide in Fig. 6 a fixpoint formulation of our first concrete interpreter restricted to the analysis of the sequential executions. The interpreter starts by initializing the kernel and the drivers, and then consuming the posted tasks. After booting the user-space applications, we use two nested fixpoint computations. The inner one consumes the posted tasks and the outer one stabilizes the effect of interrupts after firing the `sleep` event when no task is waiting.

6.2. Abstract Semantics

In this section, we present two abstraction levels for approximating the (non computable) semantics domain $\mathcal{D}$. The first abstraction level focuses on the dynamics of ADP and maintains precise information about the hardware states in order to detect forbidden transitions. While this abstraction is sound and covers every possible execution path, it may lack some precision in presence of complex control flows that use the tasks mechanisms. Therefore, we propose a second abstraction that refines the first one by adding partial information about the contents of the tasks queue in order to avoid inconsistent tasks ordering.

$$
\begin{array}{l}
\mathbf{S}[\![\texttt{X = exp}]\!]^\sharp_{\mathcal{S}} X \triangleq \\
\quad \text{let } X_1 = \mathbf{S}[\![\mathsf{event}^\star\ \texttt{X}^r]\!]^\sharp_{\mathcal{S}} X \text{ in} \\
\quad \text{let } X_2 = \lambda s.\ \mathbf{S}[\![\texttt{X = exp}]\!]^\sharp_{\mathcal{M}} \circ X_1(s) \text{ in} \\
\quad \mathbf{S}[\![\mathsf{event}^\star\ \texttt{X}^w]\!]^\sharp_{\mathcal{S}} X_2 \\
\mathbf{S}[\![\mathsf{event}^\star\ e]\!]^\sharp_{\mathcal{S}} X \triangleq \\
\quad \mathsf{lfp}\ \lambda X'.\ X' \triangledown_{\mathcal{S}} (\mathbf{S}[\![\mathsf{event}\ e]\!]^\sharp_{\mathcal{S}} X \sqcup_S \mathbf{S}[\![\mathsf{event}\ \alpha]\!]^\sharp_{\mathcal{S}} X') \\
\mathbf{S}[\![\mathsf{event}\ e]\!]^\sharp_{\mathcal{S}} X \triangleq \\
\quad \lambda s.\ \bigsqcup_{\mathcal{M}\ (s',e,s,g,a)\in\mathcal{T}} \mathbf{S}[\![a]\!]^\sharp_{\mathcal{M}} \circ \mathbf{S}[\![?g]\!]^\sharp_{\mathcal{M}} \circ X(s')
\end{array}
$$

Figure 7: Abstract transfer functions for hardware state partitioning.

$$
\begin{array}{l}
\wr \textit{Initial abstract state} \wr \\
\text{let } X_0 = \\
\quad \bot_{\mathcal{S}}[s_0 \to \mathsf{fold}\ \ (\lambda r.\ \lambda X.\ \mathbf{S}[\![r = 0]\!]^\sharp_{\mathcal{M}} X)\ \top_{\mathcal{M}\mathcal{R}}] \text{ in} \\
\wr \textit{Initialize the kernel and the drivers} \wr \\
\text{let } X_1 = \mathbf{S}[\![I_{drv}]\!]^\sharp_{\mathcal{S}} \circ \mathbf{S}[\![I_{ker}]\!]^\sharp_{\mathcal{S}} X_0 \text{ in} \\
\wr \textit{Analyze the posted tasks until emptying the queue} \wr \\
\text{let } X_2 = \mathsf{lfp}\ \lambda X. \\
\quad X \triangledown_{\mathcal{S}} (X_1 \sqcup_{\mathcal{S}} \bigsqcup_{\mathcal{S}\ t\in\mathbb{T}} \mathbf{S}[\![body(t)]\!]^\sharp_{\mathcal{S}} \circ \mathbf{S}[\![\mathsf{dequeue}\ t]\!]^\sharp_{\mathcal{S}} X) \text{ in} \\
\text{let } X_3 = \mathbf{S}[\![\mathsf{notask}]\!]^\sharp_{\mathcal{S}} X_2 \text{ in} \\
\wr \textit{Initialize the user applications} \wr \\
\text{let } X_4 = \mathbf{S}[\![I_{app}]\!]^\sharp_{\mathcal{S}} X_3 \text{ in} \\
\\
\mathsf{lfp}\ \lambda X.\ (\\
\quad \wr \textit{Analyze the posted tasks} \wr \\
\quad \text{let } X_5 = \mathsf{lfp}\ \lambda X'. \\
\quad\quad X' \triangledown_{\mathcal{S}} (X \sqcup_{\mathcal{S}} \bigsqcup_{\mathcal{S}\ t\in\mathbb{T}} \mathbf{S}[\![body(t)]\!]^\sharp_{\mathcal{S}} \circ \mathbf{S}[\![\mathsf{dequeue}\ t]\!]^\sharp_{\mathcal{S}} X') \text{ in} \\
\quad \text{let } X_6 = \mathbf{S}[\![\mathsf{notask}]\!]^\sharp_{\mathcal{S}} X_5 \text{ in} \\
\quad \wr \textit{Move the MCU to sleep mode} \wr \\
\quad \text{let } X_7 = \mathbf{S}[\![\mathsf{event}^\star\ \texttt{sleep}]\!]^\sharp_{\mathcal{S}} X_6 \text{ in} \\
\quad \wr \textit{Analyze the interrupts} \wr \\
\quad X_4 \sqcup_{\mathcal{S}} \bigsqcup_{\mathcal{S}\ i\in\mathbb{I}} \mathbf{S}[\![body(i)]\!]^\sharp_{\mathcal{S}} \circ \mathbf{S}[\![\mathsf{event}^\star\ \texttt{int}_i]\!]^\sharp_{\mathcal{S}} X_7 \\
)
\end{array}
$$

Figure 8: Abstract interpreter for sequential executions.

6.2.1. Hardware State Partitioning

To properly analyze the behaviors of a device driver, two important design goals should be considered. First, it is vital to keep accurate information about the hardware state since it is a key guidance element to correctly simulate the evolution of the ADP. Consequently, losing information about hardware state – when merging environments for example – should be avoided. Also, it is necessary to preserve some relationship between the hardware state of the ADP and the values of the registers because drivers try to infer the state of the device by inquiring its registers where state information is generally encoded in a set of bits.

Therefore, our first abstraction performs a partitioning with respect to the hardware states so that memory information about different states are not merged together. In other words, we collect the reachable memory environments separately for every hardware state s of the target ADP. Since we can not keep every possible detail about these environments, we build a sound summary of them using the *memory abstraction framework* described in Miné (2006a) and Miné (2012) that can over-approximate the effect of complex C constructs on memory variables efficiently. This abstraction framework is generic and can be used with any underlying numerical domain, such as the intervals domain presented earlier or even more complex relational domains such as octagons (Miné (2006b)) or polyhedra (Cousot and Halbwachs (1978)). However, in this work, we will limit ourself to the use of the intervals domain for its simplicity and efficiency. The details of these memory approximations are out of the scope of this paper, so we assume that we are given an abstract memory domain $\langle \mathcal{M}^\sharp, \sqsubseteq_{\mathcal{M}}, \sqcup_{\mathcal{M}}, \bot_{\mathcal{M}} \rangle$ along with a widening operator $\triangledown_{\mathcal{M}}$, a concretization function γ_M and the abstract transfer functions $\mathbf{S}[\![.]\!]^\sharp_{\mathcal{M}}$.

The formal definition of the hardware state partitioning domain $\langle \mathcal{D}^\sharp_S, \sqsubseteq_{\mathcal{S}}, \sqcup_{\mathcal{S}}, \bot_{\mathcal{S}} \rangle$ is given by:

$$\mathcal{D}^\sharp_S \triangleq \mathcal{S} \to \mathcal{M}^\sharp$$

with the following concretization function:

$$\gamma_{\mathcal{S}}(X) \triangleq \{(m, s, q) \mid q \in \mathcal{Q} \wedge s \in \mathcal{S} \wedge m \in \gamma_{\mathcal{M}} \circ X(s)\}$$

and all lattice and widening operators are defined pointwise.

Since the number $|\mathcal{S}|$ of hardware states is finite and generally small, this partitioning does not induce excessive computational costs. It is important to note that this abstraction forgets about the contents of the tasks queue which leads to a loss of precision. Indeed, without any information about the posted tasks, preserving the soundness condition implies that we must assume that the queue can be any element of $\mathcal{Q}$ which means that our analysis will compute the effect of every possible ordering of all existing tasks.

The most interesting transfer functions are presented in Fig. 7. The function $\mathbf{S}[\![\mathsf{event}\ e]\!]^\sharp_{\mathcal{S}}$ computes the abstract effect of an event e on the hardware and works by collecting for every possible next state s the set of transitions $(s', e, s, g, a) \in \mathcal{T}$ going from a previous state s' to s. The abstract memory environment at the state s' is then filtered by the guard g and transformed by the hardware assignment a. The function $\mathbf{S}[\![\mathsf{event}^\star\ e]\!]^\sharp_{\mathcal{S}}$ performs a sequence of widening-based iterations to compute an over-approximation of the effect of asynchronous events after the event e. The function $\mathbf{S}[\![\texttt{X = exp}]\!]^\sharp_{\mathcal{S}}$ is based on the previous two functions to over-approximate the effect of a register assignment on both the program and hardware state. Since we do not maintain any information about the posted tasks, the functions $\mathbf{S}[\![\mathsf{post}\ t]\!]^\sharp_{\mathcal{S}}$, $\mathbf{S}[\![\mathsf{dequeue}\ t]\!]^\sharp_{\mathcal{S}}$ and $\mathbf{S}[\![\mathsf{notask}]\!]^\sharp_{\mathcal{S}}$ are defined as the identity function.

The abstract version of the restricted interpreter for sequential executions is depicted in Fig. 8. We can notice that its structure is very similar to the concrete version, with the difference of employing the widening operator in

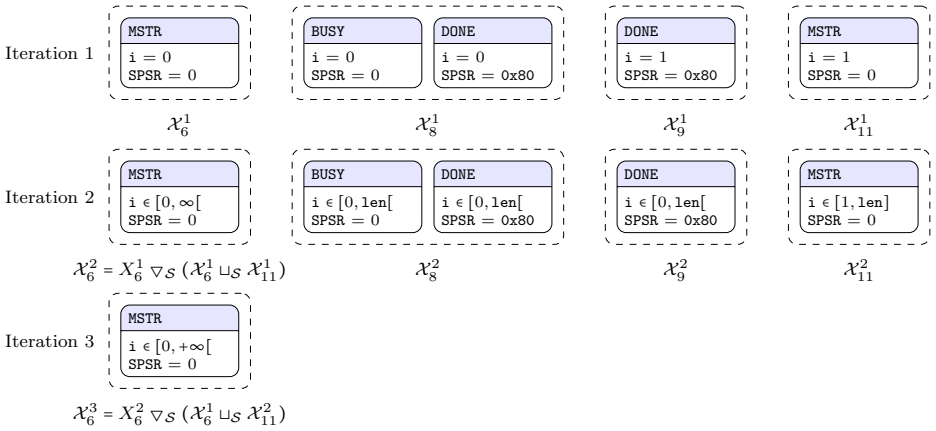


Figure 9: Results of fixpoint iterations obtained during the analysis of the task-less SPI driver using hardware state partitioning.

order to accelerate the convergence of the fixpoint iterations for consuming tasks and firing interrupts.

Example 5. To explain the intuition behind this first abstraction, let us consider again the ADP of the SPI subsystem and its driver example presented in Fig. 3 and 4(a) respectively. The main steps of the analysis iterations are presented in Fig. 9 where we use the notation $\mathcal{X}_l^i$ to denote the abstract environment at line l during iteration i.

When the execution reaches for the first time the while loop at line 6, the ADP is in state `MSTR`. After the assignment statement at line 7 modifies the `SPDR` data register, the ADP moves to state `BUSY`. Since the SPI communication is asynchronous, the ADP can change its state to `DONE` at any moment, which is expressed in the abstract state $\mathcal{X}_8^1$ by two distinct state partitions. It is important to note that the value of the status register `SPSR` is different between these two partitions. This disjunction allows the analysis to infer the correct abstract environment $\mathcal{X}_9^1$ that indicates that the ADP should be in state `DONE` after the polling loop.

When performing the second fixpoint iteration of the while loop at line 6, the value of `i` is extrapolated to $[0, +\infty[$ using the widening operator. The same previous behavior is observed: the ADP moves to states `BUSY` or `DONE` depending on the termination of the transfer and the polling loop will discard the first state partition by filtering on the value of the status register `SPSR`.

At the end of the program, the `BUG` state was not reached at any step of the analysis, which constitutes a proof that the property is not violated. □

6.2.2. Tasks Queue Partitioning

In some TinyOS drivers, the control flow of hardware interactions is implemented by tasks in order to free up the scheduler during polling periods. In such situations, the previous abstract domain $\mathcal{D}_{\mathcal{S}}^{\sharp}$ is too imprecise to reconstruct the real control flow since no information is maintained by $\mathcal{D}_{\mathcal{S}}^{\sharp}$ about the tasks queue. Therefore, it is necessary to refine the previous abstraction to preserve a partial view on the contents of the queue. An efficient solution is to count the number of occurrences of every task and ignore their order in the queue. This abstraction is known as a Parikh vector (Parikh (1966)) which is generally used to approximate sets of sequences/collections of discrete objects (Feret (2001)). Unfortunately, this technique is not sufficient to prove the correctness of several task-based drivers, as we illustrate in the following example.

Example 6. We consider the previous task-based SPI driver presented in Fig. 4(b). An analysis with the state partitioning domain presented in the previous section will fatally lead to the `BUG` state. Indeed, since no information is available about posted tasks, the task `tx` can be executed in the initial hardware state `OFF`, resulting in a forbidden data transfer over the inactivated SPI bus.

Even if we extend $\mathcal{D}_{\mathcal{S}}^{\sharp}$ with a Parikh vector, the analysis can not eliminate the false alarm. To explain further this problem, we depict in Fig. 10 the results of the fixpoint iterations using the extended domain and we focus on the obtained abstract environments during the tasks consumption step, which corresponds to the computation of X_5 in the previous abstract interpreter. The execution trace of these iterations can be summarized as follows:

1. Initially, after executing the `send_spi` function, the task `tx` is posted to send the first byte while the ADP is in state `MSTR`.
2. During the first fixpoint iteration, the task `tx` is executed and the transfer is started, which generates two additional partitions for the states `BUSY` and `DONE`. The task `check` is posted in order to continuously check the termination of the transfer.
3. The task `check` filters its input abstract environment depending on the status flag `SPIF`. The next byte is prepared to be sent when this flag is set, which corresponds to the state partition `DONE`. Since we are using the widening operator, the byte index `i` is extrapolated to $[0, +\infty[$.

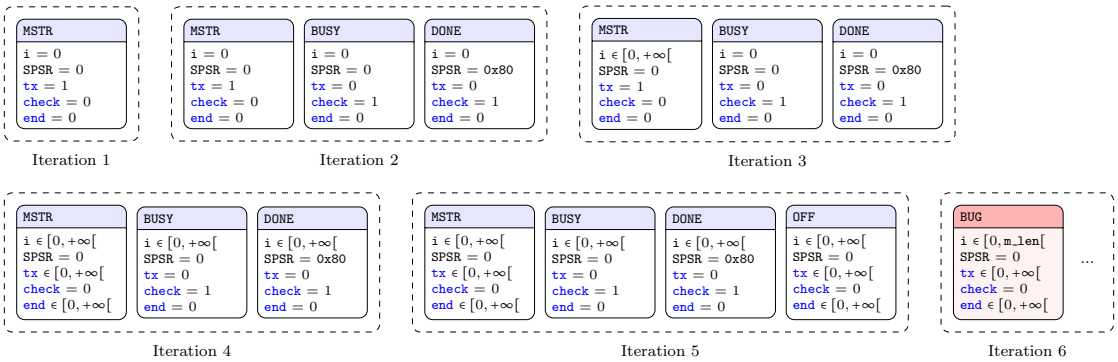


Figure 10: The analysis trace of a false positive during the analysis of the task-based SPI driver using the hardware state partitioning.

4. The execution of the task `tx` over this new abstract environment will generate two cases: the transfer of the next byte or the end of the transfer. The latter case is performed by posting the task `end`. It is important to note that this task is posted while the ADP is in state `MSTR`. By merging this result with the previous `MSTR` partition, and by widening, we reach an imprecise and nondeterministic situation resulting from the fact that `tx` $\in [0, +\infty[$ and `end` $\in [0, +\infty[$. These values imply that no constraint is available for these two tasks, and therefore any ordering of them can happen.
5. When the analyzer executes the task `end` in this imprecise context, it will shut down the SPI, moving the ADP to state `OFF`. Due to the previous nondeterminism, the task `tx` can be executed in this hardware state, which will lead to a false positive after writing to data register `SPDR` while the bus is inactive.

□

The previous example shows that counting the number of posts can be insufficient to reconstitute a correct execution flow of tasks. More precisely, the false positive originated from a lack of relation between the entries of the Parikh vector. Indeed, for this illustrative example, there exists an exclusive-or relation between the presence of the tasks `tx` and `end` together in the queue, and providing the analyzer with such information will eliminate the false positive. To provide such relationship, we propose to build a partitioning with respect to the presence of tasks. This form of disjunction allows the analyzer to keep separate two sets of environments when a task is not posted in both cases.

Formally, we define the task partitioning abstract domain $\langle \mathcal{D}^\sharp_{\mathcal{Q}}, \sqsubseteq_{\mathcal{Q}}, \sqcup_{\mathcal{Q}}, \bot_{\mathcal{Q}} \rangle$ having the following structure:

$$\mathcal{D}^\sharp_{\mathcal{Q}} \triangleq \wp(\mathbb{T}) \to (\mathcal{D}^\sharp_{\mathcal{S}} \times (\mathbb{T} \to \mathcal{N}^\sharp))$$

where $\langle \mathcal{N}^\sharp, \sqsubseteq_{\mathcal{N}}, \sqcup_{\mathcal{N}}, \bot_{\mathcal{N}} \rangle$ is a numeric abstract domain approximating a set of integers, such as intervals, provided with its concretization function $\gamma_{\mathcal{N}}$. Basically, when $X \in \mathcal{D}^\sharp_{\mathcal{Q}}$ is an abstract environment and $T \in \wp(\mathbb{T})$ is a set of tasks, the partition $X(T)$ provides an over-approximation of the memory, hardware and queue environments when only the tasks $t \in T$ are present in the queue. To obtain the concrete environments approximated by the abstract environment X, we define the following concretization function:

$$\begin{aligned}
&\gamma_{\mathcal{Q}}(X) \triangleq \\
&\quad \{(m, s, q) \mid \exists T \in \wp(\mathbb{T}) : (m, s, -) \in \gamma_{\mathcal{S}} \circ \downarrow_S \circ X(T) \\
&\qquad \wedge \forall t \in T : \\
&\qquad\qquad \mathtt{count}(q, t) \in \gamma_{\mathcal{N}} \circ \downarrow^t_Q \circ X(T) \wedge \mathtt{count}(q, t) > 0 \\
&\qquad \wedge \forall t \notin T : \mathtt{count}(q, t) = 0\}
\end{aligned}$$

where $\downarrow_S (X_S, X_T)$, $\downarrow_Q (X_S, X_T)$ and $\downarrow^t_Q (X_S, X_T)$ are three projection operators to retrieve respectively X_S, X_T and $X_T(t)$ from an abstract environment $(X_S, X_T) \in \mathcal{D}^\sharp_{\mathcal{Q}}$. Intuitively, the function $\gamma_{\mathcal{Q}}$ obtains the concrete memory and hardware environments using the previous concretization function $\gamma_{\mathcal{S}}$. The concrete queues environments are constructed using the condition that the number of occurrences of every task should be in the range of its corresponding entry in the Parikh vector.

Since the transfer functions for this domains share many similar constructs, we limit ourselves to the presentation of the case of the statement post t:

$$\begin{aligned}
&\mathbf{S}[\![\mathsf{post}\, t]\!]^\sharp_{\mathcal{Q}} X \triangleq \\
&\quad \lambda T. \begin{cases} \bot_{\mathcal{S}}, \lambda t'.\ \bot_{\mathcal{N}} \text{ if } t \notin T \\ \downarrow_S \circ X(T), \\ \quad \downarrow_Q \circ X(T)[t \leftarrow \mathtt{inc}^\sharp_{\mathcal{N}} \circ \downarrow^t_Q \circ X(T)] \text{ otherwise} \end{cases} \\
&\quad \sqcup_{\mathcal{Q}} \\
&\quad \lambda T. \begin{cases} \bot_{\mathcal{S}}, \lambda t'.\ \bot_{\mathcal{N}} \text{ if } t \notin T \\ \downarrow_S \circ X(T \setminus \{t\}), \\ \quad \downarrow_Q \circ X(T \setminus \{t\})[t \leftarrow \mathtt{one}^\sharp_{\mathcal{N}}] \text{ otherwise} \end{cases}
\end{aligned}$$

The first part of the union $\sqcup_{\mathcal{Q}}$ handles the case where the task t was already posted and operates by incrementing the number of its occurrences using the abstract incrementation function $\mathtt{inc}^\sharp_{\mathcal{N}}$ that verifies the soundness condition:

$$\forall X \in \mathcal{N}^\sharp : \gamma_{\mathcal{N}} \circ \mathtt{inc}^\sharp_{\mathcal{N}}(X) \supseteq \{i + 1 \mid i \in \gamma_{\mathcal{N}}(X)\}$$

The case where t was not present in the queue is handled by the second part, which updates the partitions $X(T \setminus \{t\})$ by setting the Parikh vector entry of t to the abstract

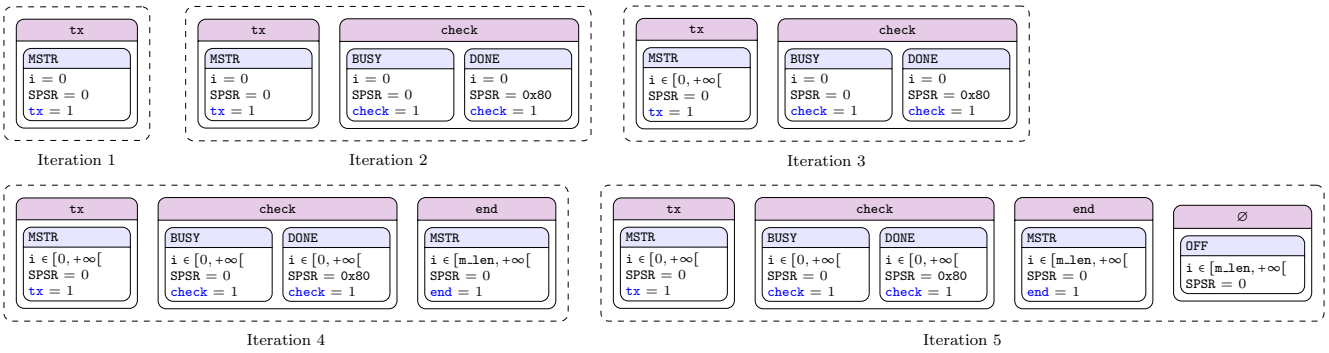


Figure 11: A summary of fixpoint iterations obtained during the analysis of the SPI driver using tasks queue partitioning.

element $\texttt{one}^\sharp_{\mathcal{N}}$ that verifies the soundness condition: $1 \in \gamma_{\mathcal{N}}(\texttt{one}^\sharp_{\mathcal{N}})$.

Example 7. We illustrate in Fig. 11 the advantage of the tasks queue partitioning for proving the correctness of the previous SPI driver. The execution trace is relatively similar to the previous case for the first three iterations. During the fourth iteration, the task `end` is posted in a partition different from the task `tx`, which avoids the previous nondeterminism and shuts down the SPI bus safely.

7. Preemptive Executions Analysis

Until now, we have considered that interrupts can occur only during inactivity periods of the MCU. This assumption allowed us to simplify the presentation of the abstractions related to the hardware state and the tasks queue. In this section, we extend the previous analysis to take into consideration the preemption of execution by interrupts at any moment of the program lifetime. We define new concrete and abstract semantics, that build on the previous ones, to soundly over-approximate the set of reachable hardware states during all possible concurrent executions.

7.1. Concrete Semantics

To add interrupts preemption to our previous analysis, we need to care about (*i*) when an interrupt can be fired and (*ii*) when the MCU is configured to execute its corresponding interrupt vector. In this work, we focus on the second consideration and we approximate the first one using nondeterminism. In other words, as long as an interrupt is not *masked* by software, we consider that it can happen at any moment, which can be implemented as a nondeterministic choice to execute or not the interrupt handler before executing any statement. Nevertheless, the imprecision caused by the nondeterminism can be reduced by filtering the hardware states in which interrupts can not occur. This can be done by adding transitions, labeled with interrupt events $\texttt{int}_i$, that go from the filtered states to a special absorbing state that has no successor.

An interrupt can be masked at two levels: *globally* and *partially*. The first level is handled by a Global Interrupt Enable (GIE) bit found in most MCUs. For the case of the ATmega128 MCU that we are considering in this work, the I bit located at the last position in the status register `SREG` must be set in order to enable interrupts. In the following, we will denote by gcond $\triangleq$ `SREG & (1 << 7) != 0` the condition expression that verifies that the I bit is set in `SREG`. Also, we define two shortcut statements `cli` $\triangleq$ `SREG &= ~(1 << 7)` and `sei` $\triangleq$ `SREG |= (1 << 7)` to respectively clear and set the I bit.

The second masking level consists in the inhibition of a partial set of interrupts, performed generally through the configuration of particular control registers. Since these configurations differ from an interrupt to another, we define the function icond $\in \mathbb{I} \to Stmt_C$ giving for every interrupt its corresponding firing condition, formulated as a C boolean expression. For example, to allow the occurrence of the Timer0 compare interrupt (T0CI), the following condition should be verified:

```
icond(T0CI) ≜  (TCCR0 & ((1 << CS02) | (1 << CS01) |
                  (1 << CS00)))
               && (TIMSK & (1 << TOIE0))
```

The first condition ensures that a non null prescaler is configured in the control register `TCCR0`, otherwise no clock will source the timer sub-system. The second condition checks whether this particular interrupt is enabled in the timer mask register `TIMSK`.

Since these conditions are expressed as predicates over hardware registers – which are considered as normal program variables – we can use our previous concrete semantic domain $\mathcal{D}_{\mathcal{I}} \triangleq \mathcal{D}$ to encapsulate the mask values of interrupts. However, we need to extend the transfer functions in order to handle the nondeterministic execution of interrupt vectors when they are not masked. To do so, we only need to define the transfer functions for the atomic C statements (assignments and tests) and for the additional statements set $Stmt_H$ and $Stmt_Q$. The remaining transfer functions are kept unmodified because they are ultimately reduced, by structural induction on the syntax, to atomic statements. Let s be one of these atomic statements. We

define its preemption-aware transfer function as follows:

$$\begin{aligned}&\mathbf{S}[\![s]\!]_{\mathcal{I}}R \triangleq \mathbf{S}[\![s]\!](R \cup \bigcup_{i\in\mathbb{I}}\\ &\quad \text{let } R_1 = \mathbf{S}[\![\mathsf{event}^\star\ \mathtt{int}_i]\!] \circ \mathbf{S}[\![?\,\mathsf{icond}(i)]\!] \circ \mathbf{S}[\![?\,\mathsf{gcond}]\!]R \text{ in}\\ &\quad \mathbf{S}[\![\mathtt{sei}]\!] \circ \mathbf{S}[\![\mathit{body}(i)]\!]_{\mathcal{I}} \circ \mathbf{S}[\![\mathtt{cli}]\!]R_1)\end{aligned}$$

Basically, this function executes the statement s over the union of the input environments R and the post-execution environments of the enabled interrupts. These environments are obtained by first filtering R in order to keep only the environments where the condition expressions of the global enable bit and the partial mask of i are true. After that, we signal the occurrence of the interrupt to the ADP by calling the function $\mathbf{S}[\![\mathsf{event}^\star\ \mathtt{int}_i]\!]$. Finally, we execute the interrupt vector body by first clearing the global interrupt enable bit and then setting it again as specified by the ATmega128 data sheet.

Two important points should be noted. First, using the union of the post-interrupts environments implies that *at most* one interrupt handler is executed. This choice is justified by the fact that the MCU, after returning from an interrupt, must execute at least one instruction before allowing the execution of the next interrupt, which avoids a continuous succession of interrupts that prevents a function from terminating its computations. Second, this formulation allows the analysis of nested interrupts by using the preemptive transfer function $\mathbf{S}[\![\mathit{body}(i)]\!]_{\mathcal{I}}$. Nevertheless, since the I-bit is automatically cleared at the beginning of the interrupt, the body of the corresponding routine should execute the `sei` statement to allow this feature, which is taken into account by our semantics.

Using this preemption mechanism, we define in Fig. 12 the concrete preemptive interpreter of TinyOS programs. It shares with the previous sequential interpreter, presented in Fig. 6, most of its structure with two major differences. Firstly, the body of the initialization procedures, tasks and interrupts are analyzed using the preemption-aware transfer function $\mathbf{S}[\![.]\!]_{\mathcal{I}}$, instead of the sequential version $\mathbf{S}[\![.]\!]$. Secondly, we have instrumented the interpreter with statements to control the global interrupt mask as performed by the TinyOS scheduler. Note that, at specific locations, the scheduler saves the register `SREG` in a backup variable, that we denote by `oldSREG`, in order to restore it later to preserve the modifications performed by the tasks.

7.2. Abstract Semantics

In this section, we develop an abstraction of the domain $\mathcal{D}_{\mathcal{I}}$ and the transfer functions $\mathbf{S}[\![.]\!]_{\mathcal{I}}$ that approximate the dynamics of preemptive executions. To do so, we divide our problem into two parts: (*i*) the maintenance of an abstract view about the enabled interrupts, and (*ii*) the computation of the effect of an enabled interrupt on the execution flow. Therefore, we start by presenting an approximation of the masking system before describing how to use this abstraction to analyze the preemptive executions of a TinyOS program.

$$\begin{aligned}&\text{let } R_0 = \{(m, s_0, \varnothing_{\mathcal{Q}}) \mid\\ &\quad m \in \mathsf{fold}\ (\lambda r.\lambda R.\ \mathbf{S}[\![r = 0]\!]_{\mathcal{M}}R)\ \mathcal{M}\ \mathcal{R}\} \text{ in}\\ &\text{let } R_1 = \mathbf{S}[\![I_{drv}]\!]_{\mathcal{I}} \circ \mathbf{S}[\![I_{ker}]\!]_{\mathcal{I}}R_0 \text{ in}\\ &\text{let } R_2 = \mathsf{lfp}\ \lambda R.\ R_1 \cup \bigcup_{t\in\mathbb{T}} \mathbf{S}[\![\mathit{body}(t)]\!]_{\mathcal{I}} \circ \mathbf{S}[\![\mathsf{dequeue}\,t]\!]_{\mathcal{I}}R \text{ in}\\ &\text{let } R_3 = \mathbf{S}[\![\mathsf{notask}]\!]_{\mathcal{I}}R_2 \text{ in}\\ &\text{let } R_4 = \mathbf{S}[\![\mathtt{oldSREG = SREG}]\!] \circ \mathbf{S}[\![I_{app}]\!]_{\mathcal{I}} \circ \mathbf{S}[\![\mathtt{sei}]\!]R_3 \text{ in}\\ &\\ &\mathsf{lfp}\ \lambda R.\ (\\ &\quad \text{let } R_5 = \mathbf{S}[\![\mathtt{SREG = oldSREG}]\!]R \text{ in}\\ &\quad \text{let } R_6 =\\ &\qquad \mathsf{lfp}\ \lambda R'.\ R_5 \cup \bigcup_{t\in\mathbb{T}} \mathbf{S}[\![\mathit{body}(t)]\!]_{\mathcal{I}} \circ \mathbf{S}[\![\mathsf{dequeue}\,t]\!]_{\mathcal{I}}R' \text{ in}\\ &\quad \text{let } R_7 = \mathbf{S}[\![\mathsf{notask}]\!]_{\mathcal{I}}R_6 \text{ in}\\ &\quad \text{let } R_8 = \mathbf{S}[\![\mathtt{sei}]\!] \circ \mathbf{S}[\![\mathtt{oldSREG = SREG}]\!]R_7\\ &\quad \text{let } R_9 = \mathbf{S}[\![\mathsf{event}^\star\ \mathtt{sleep}]\!]R_7 \text{ in}\\ &\quad R_4 \cup\\ &\qquad \bigcup_{i\in\mathbb{I}} \mathbf{S}[\![\mathtt{sei}]\!] \circ \mathbf{S}[\![\mathit{body}(i)]\!]_{\mathcal{I}} \circ \mathbf{S}[\![\mathtt{cli}]\!] \circ \mathbf{S}[\![\mathsf{event}^\star\ \mathtt{int}_i]\!]R_9\\ &)\end{aligned}$$

Figure 12: Concrete interpreter for preemptive executions.

7.2.1. Abstraction of Interrupt Masks

We approximate the interrupt masks by breaking the relation between the global masking level and the partial one. This separation allows us to build efficient transfer functions by sacrificing some precision. Formally, we define the abstract mask domain $\langle \mathcal{D}^\sharp_{\mathcal{K}}, \sqsubseteq_{\mathcal{K}}, \sqcup_{\mathcal{K}}, \bot_{\mathcal{K}} \rangle$ as the following product:

$$\mathcal{D}^\sharp_{\mathcal{K}} \triangleq \underbrace{\{\bot_{\mathbb{I}}, \mathbb{0}, \mathbb{1}, \top_{\mathbb{I}}\}}_{\mathcal{K}^\sharp_{\mathcal{G}}} \times \underbrace{(\wp(\mathbb{I}) \to \mathcal{D}^\sharp_{\mathcal{Q}})}_{\mathcal{K}^\sharp_{\mathcal{P}}}$$

The global mask is maintained by the lattice $\mathcal{K}^\sharp_{\mathcal{G}}$ that is identical to the powerset lattice $\wp(\{0,1\})$ and encodes all possible states of the I bit. The second masking level is provided by the lattice $\mathcal{K}^\sharp_{\mathcal{P}}$ defining a partitioning with respect to the activated interrupts. From the pointwise definition of the lattice operators of $\mathcal{K}^\sharp_{\mathcal{P}}$ and the simple definition of $\mathcal{K}^\sharp_{\mathcal{G}}$, we can easily derive the definition of the bottom element $\bot_{\mathcal{K}}$ and the operators $\sqcup_{\mathcal{K}}$, $\sqsubseteq_{\mathcal{K}}$ and $\triangledown_{\mathcal{K}}$.

The set of concrete environments corresponding to a given abstract interrupt mask is given by the following

$$
\begin{array}{l}
\mathbf{S}[\![\texttt{SREG = exp}]\!]^\sharp_{\mathcal{K}}(G,X) \triangleq \\
\quad \text{let } X_1 = \lambda I.\ \mathbf{S}[\![\texttt{SREG = exp}]\!]^\sharp_{\mathcal{Q}} \circ X(I) \text{ in} \\
\quad \text{let } G_1 = \begin{cases}
\mathbb{1} & \text{if } \forall I \in \wp(\mathbb{I}) : \mathbf{S}[\![?\,\mathsf{gcond}]\!]^\sharp_{\mathcal{Q}} \circ X_1(I) \neq \bot_{\mathcal{Q}} \wedge \mathbf{S}[\![?!\,\mathsf{gcond}]\!]^\sharp_{\mathcal{Q}} \circ X_1(I) = \bot_{\mathcal{Q}} \\
\mathbb{0} & \text{if } \forall I \in \wp(\mathbb{I}) : \mathbf{S}[\![?\,\mathsf{gcond}]\!]^\sharp_{\mathcal{Q}} \circ X_1(I) = \bot_{\mathcal{Q}} \wedge \mathbf{S}[\![?!\,\mathsf{gcond}]\!]^\sharp_{\mathcal{Q}} \circ X_1(I) \neq \bot_{\mathcal{Q}} \\
\top_{0\mathbb{I}} & \text{if } \forall I \in \wp(\mathbb{I}) : \mathbf{S}[\![?\,\mathsf{gcond}]\!]^\sharp_{\mathcal{Q}} \circ X_1(I) \neq \bot_{\mathcal{Q}} \wedge \mathbf{S}[\![?!\,\mathsf{gcond}]\!]^\sharp_{\mathcal{Q}} \circ X_1(I) \neq \bot_{\mathcal{Q}} \\
\bot_{0\mathbb{I}} & \text{otherwise}
\end{cases} \\
\quad \text{in } (G_1, X_1)
\end{array}
$$

$$
\begin{array}{l}
\mathbf{S}[\![\texttt{X = exp}]\!]^\sharp_{\mathcal{K}}(G,X) \triangleq \\
\quad \text{let } X_1 = \lambda I.\ \mathbf{S}[\![\texttt{X = exp}]\!]^\sharp_{\mathcal{Q}} \circ X(I) \text{ in} \\
\quad \text{let } X_2 = \lambda I.\ \bigsqcup_{\mathcal{Q}\ i \in I, I' \in \wp(\mathbb{I})} \mathbf{S}[\![?\,\mathsf{icond}(i)]\!]^\sharp_{\mathcal{Q}} \circ X_1(I') \sqcup_{\mathcal{Q}} \bigsqcup_{\mathcal{Q}\ i \notin I, I' \in \wp(\mathbb{I})} \mathbf{S}[\![?!\,\mathsf{icond}(i)]\!]^\sharp_{\mathcal{Q}} \circ X_1(I') \\
\quad \text{in } (G, X_2)
\end{array}
$$

Figure 13: Abstract transfer functions for the $\mathcal{D}^\sharp_{\mathcal{K}}$ domain.

concretization function:

$$
\begin{array}{l}
\gamma_{\mathcal{K}}(G,X) \triangleq \\
\quad \text{let } R = \{(m,s,q) \mid \exists I \in \wp(\mathbb{I}) : (m,s,q) \in \gamma_{\mathcal{Q}} \circ X(I) \wedge \\
\qquad \forall i \in I : (m,s,q) \in \mathbf{S}[\![?\,\mathsf{icond}(i)]\!] \circ \gamma_{\mathcal{Q}} \circ X(I) \wedge \\
\qquad \forall i \notin I : (m,s,q) \in \mathbf{S}[\![?!\,\mathsf{icond}(i)]\!] \circ \gamma_{\mathcal{Q}} \circ X(I)\} \\
\quad \text{in match } G \text{ with} \\
\qquad \mid \top_{0\mathbb{I}} \to R \\
\qquad \mid \mathbb{1} \to \mathbf{S}[\![?\,\mathsf{gcond}]\!]R \\
\qquad \mid \mathbb{0} \to \mathbf{S}[\![?!\,\mathsf{gcond}]\!]R \\
\qquad \mid \bot_{0\mathbb{I}} \to \varnothing
\end{array}
$$

Let us define now the abstract transfer functions $\mathbf{S}[\![.]\!]^\sharp_{\mathcal{K}}$ related to $\mathcal{D}^\sharp_{\mathcal{K}}$. The most important cases are presented in Fig. 13. The transfer function $\mathbf{S}[\![\texttt{SREG = exp}]\!]^\sharp_{\mathcal{K}}(G,X)$ handles the change of the global enable bit I after a modification of the `SREG` status register, as performed by the two shortcut statements `sei` and `cli`. After updating each partition with the assignment statement, we check the different values of the mask expression gcond over the resulting environments and update the abstract global bit accordingly. The effect of changing the partial masks is defined by the function $\mathbf{S}[\![\texttt{X = exp}]\!]^\sharp_{\mathcal{K}}(G,X)$, where `X` is a hardware register different than `SREG` and present in at least one of the expressions icond. The function updates the partitions with the effect of the assignment and rebuilds the partitions again depending on the evaluation of the expressions icond.

7.2.2. Abstraction of Preemption

Dealing with preemption in interrupt-rich programs is a challenging task. Several approaches have been developed offering different precision/efficiency tradeoffs. Sequentialization (Monniaux (2007); Bucur and Kwiatkowska (2011)) is a simple solution consisting in instrumenting the original program with nondeterministic calls to the interrupt handlers. Since the number of execution paths may become intractable, different forms of partial order reduction are proposed to restrict the locations of this instrumentation. This method allows a precise analysis, but becomes inefficient in the presence of a large number of interrupts with possible nested occurrences. A more interesting approach, proposed i-CBMC model checker (Kroening et al. (2015)), alleviates the need to apply partial order reductions and provides a better scalability with less instrumentation effort. It is based on the definition of a partial order on preemption traces that uses a set of logical clocks to symbolically encode the different interleavings of interrupts. Whilst this method is effective in many test cases, it lacks the soundness guarantee and can not cover all possible execution traces of complex programs in finite time.

In our work, we aim at proposing a more efficient approach that guarantees the soundness condition and avoids executing interrupt handlers every time they are enabled. Our solution is based on the Modular Abstract Interpretation framework (Cousot and Cousot (2002)) and is similar to the approach of the static analyzer AstréeA (Miné (2011, 2014)). The general idea of this method consists in analyzing the *parts* of the program *separately* and then compose the local results of every part to get an aggregate view of the whole program. Since the interactions between these parts can be complex, the analysis may be iterated several times to obtain the correct results. Indeed, the initial analysis iteration has no information about the influence of a part on another and is therefore performed by assuming that there is no such interactions. However, during this iteration, the analysis can discover new interactions *on the fly*, such as new call sites, providing a more accurate view on the actual interaction map. Consequently, successive iterations are required until we ensure that all interactions have been discovered.

In our case, the high-level functions (initialization functions, tasks and interrupts) constitute the *parts* of the program with the restriction that only interrupts can preempt execution. The analysis processes each part separately and constructs two inter-parts information: the preemption contexts and the return contexts:

1. The *preemption context* of an interrupt represents the collection of the abstract environments where an interrupt may occur. It is constructed on the fly during the analysis of the other parts by computing the union

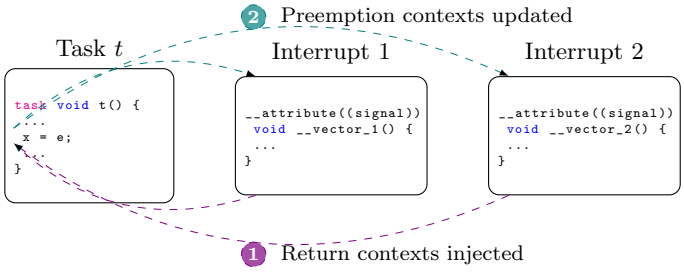


Figure 14: Approximation of interrupts preemption using the return contexts of interrupts. Preemption contexts are updated for an eventual next iteration.

of the abstract environments that verify the enable condition of the interrupt.

2. The analysis computes the *return contexts* of interrupts by executing separately each interrupt handler over its corresponding preemption context. It will be used during the next iteration to soundly *emulate* the execution of the interrupt handler whenever the interrupt is enabled.

An illustrative example of this mechanism is depicted in Fig. 14 where a task t is analyzed with two preempting interrupts. Let us denote by X the abstract environment reaching the statement s : `x = e`. The approximation of the eventual preemption before s is performed by merging X with the return contexts of the enabled interrupts before passing the resulting environment to the transfer function of the statement. In addition, if X contains new state information not present in the current preemption contexts, the latter should be updated in order to perform a new iteration that will compute the new return contexts that consider these modifications.

Formally, we define our preemptive abstract domain $\mathcal{D}^\sharp_\mathcal{I}$ as the following product:

$$\mathcal{D}^\sharp_\mathcal{I} \triangleq \mathcal{D}^\sharp_\mathcal{K} \times (\mathbb{I} \to \mathcal{D}^\sharp_\mathcal{K}) \times (\mathbb{I} \to \mathcal{D}^\sharp_\mathcal{K})$$

The first element of this product corresponds to the abstract environment of the current flow over which statements are executed. The second and the third elements represent two maps giving for every interrupt its preemption and return contexts respectively. The definitions of $\bot_\mathcal{I}$, $\sqsubseteq_\mathcal{I}$, $\sqcup_\mathcal{I}$ and $\triangledown_\mathcal{I}$ are easily derived from this definition. For an atomic statement s, we define its abstract transfer function as:

$$\begin{array}{l}
\mathbf{S}[\![s]\!]^\sharp_\mathcal{I}(X_c, X_p, X_r) \triangleq \\
\quad \text{let } I_{en} = \{i \in \mathbb{I} \mid \mathbf{S}[\![?\,\mathsf{icond}(i)]\!]^\sharp_\mathcal{K} \circ \mathbf{S}[\![?\,\mathsf{gcond}]\!]^\sharp_\mathcal{K} X_c \neq \bot_\mathcal{K}\} \text{ in} \\
\quad \text{let } X'_c = \mathbf{S}[\![?!\,\mathsf{gcond}]\!]^\sharp_\mathcal{K} X_c \sqcup_\mathcal{K} \bigsqcup_{\mathcal{K}, i \in I_{en}} X_r(i) \text{ in} \\
\quad \text{let } X'_p = \lambda i. \begin{cases} X_c \sqcup_\mathcal{K} X_p(i) & \text{if } i \in I_{en} \\ X_p(i) & \text{otherwise} \end{cases} \text{ in} \\
\quad (\mathbf{S}[\![s]\!]^\sharp_\mathcal{K} X'_c, X'_p, X_r)
\end{array}$$

The intuition behind this definition can be explained as follows. First, we construct the set I_{en} of enabled interrupts using the enable mask expressions. After that, we merge their return contexts with the current abstract environment X_c to over-approximate the effect of the nondeterministic preemption. Also, we update the preemption contexts of the enabled interrupts with X_c. Finally, we apply the statement abstract transfer function $\mathbf{S}[\![s]\!]^\sharp_\mathcal{K}$ on the newly computed environment X'_c.

Our modular abstract interpreter for the analysis of preemptive executions is presented in Fig. 15 and operates as follows. Given the input preemption and return contexts X_p and X_r, we execute the *main* TinyOS program, which consists in executing the different initialization procedures and then entering the infinite tasks-sleep-interrupt loop, but without executing the interrupt handlers. During the analysis of the main program, the functions $\mathbf{S}[\![.]\!]^\sharp_\mathcal{I}$ collect the preemption contexts of every interrupt. After reaching the fixpoint of the infinite loop, we execute every interrupt on its preemption context. As for the main program, we also collect during the analysis the preemption contexts of every interrupt. To compute the new preemptive and return contexts $X'_p(i), X'_r(i)$ of an interrupt i for the next iteration, we proceed as follows. For $X'_r(i)$, we just retrieve the post-execution environment reached at the end of the analysis of the vector of interrupt i. For the preemption context $X'_p(i)$, we merge the environments $X^{main}_p(i)$ collected during the analysis of the main program with those collected during the analysis of other interrupts. Note that this formulation assumes that there is no reentrant interrupt, i.e. an interrupt does not allow, during its execution, being interrupted by itself. Finally, to accelerate the convergence of the fixpoint computation, we use the widening operator when computing the new preemption contexts $X'_p(i)$.

8. Experiments

In this section we describe the experimental results of the analysis of our motivating example and other real-world TinyOS device drivers using a prototype of our analysis called SADA (*Static Analyzer with Device Abstraction*) that supports both sequential and preemptive execution models. We implemented SADA using the OCaml language. The implementation consists of 4.000 lines of code and uses the CIL framework (Necula et al. (2002)) for parsing the input C files generated by the `ncc` compiler. It also builds upon the Apron library developed by Jeannet and Miné (2009) that provides a rich collection of numerical abstract domains, such as intervals, octagons and polyhedra. For our experiments, we used the interval domain enriched with modular arithmetics operations to handle the finite-size representation of numbers.

To assess the efficiency and precision of SADA, we first analyzed some device drivers of the ATmega128 MCU from the latest TinyOS 1.x release. We chose three test cases with growing complexities, in terms of lines of codes and the tasks/interrupts execution flows. For each case, a set of ADPs were verified and we were interested in three metrics: the analysis time, the peak memory consumption and

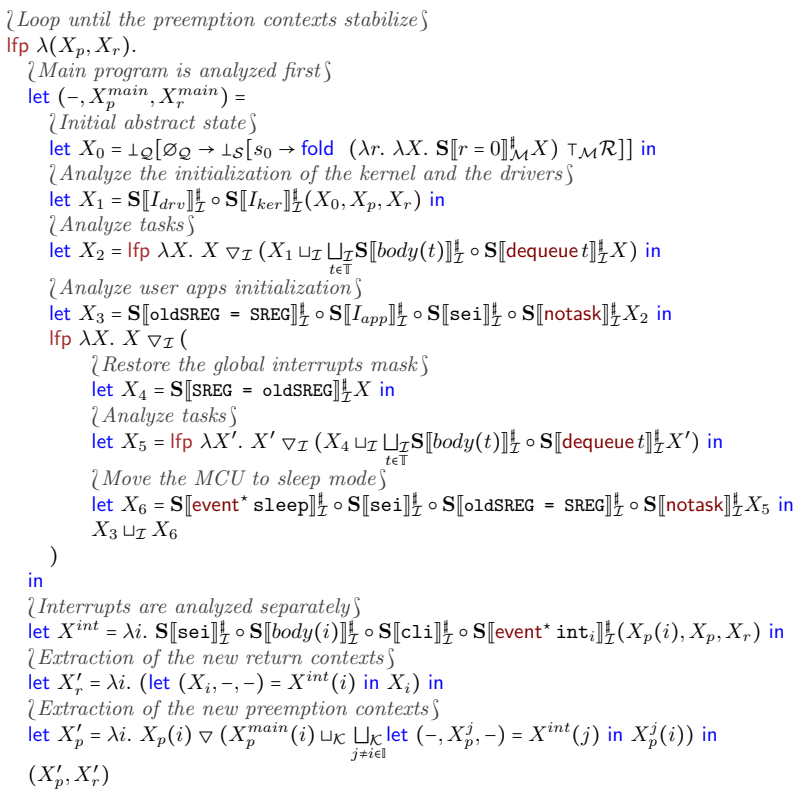


Figure 15: Abstract interpreter for preemptive executions.

the nature of the reported alarms. In total, seven ADPs were analyzed that capture the most recurrent programming patterns in embedded device driver development.

The second set of experiments consists in the analysis of the same ADPs but on a different implementation, that is, on the version 2.x of TinyOS. It is worth noting that TinyOS 2.x has been completely re-written with drastic changes in the design and the implementation. Therefore, analyzing different versions of the same driver, while keeping the ADPs unchanged, allows showing the capacity of the tool in analyzing the same specifications but on several, and possibly extremely different, implementations without additional effort from the user to configure the tool or accommodate the source code.

Finally, we also run these experiments using i-CBMC in order to compare its performances to SADA. The reason behind this choice is that it is the most efficient state-of-the-art analysis tool available for interrupt-based programs (Kroening et al. (2015)). However, we limited the use of i-CBMC to the analysis of TinyOS 2.x device drivers only because using i-CBMC requires a considerable amount of time in instrumenting the source for emulating the asynchronous hardware operations and the arrival of interrupts, as explained later in this section.

8.1. Test Cases

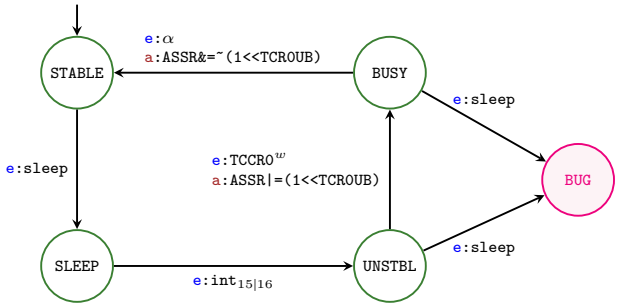
Embedded device driver development shares many programming practices that can be applied to different hardware architectures. Polling on status bits, interrupt-based serial transfer and GPIO configurations are some examples of frequent patterns that represent important building blocks in most implementations of device drivers. In this section, we briefly describe some instantiations of these recurrent patterns on the ATmega128 platform, along with some functional properties for ensuring their correctness.

8.1.1. Asynchronous Timer

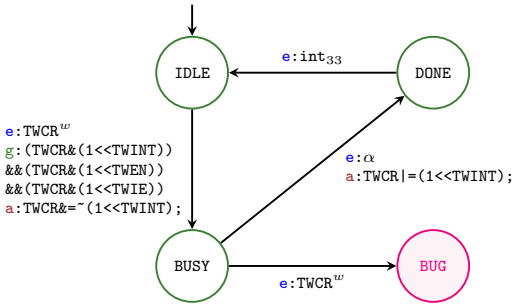
The ATmgega128 provides four hardware timers with different capabilities and applications. The 8-bit Timer/-Counter0 is frequently used in low-power embedded applications since it is the only timer that allows going to a deep sleep mode (in terms of energy consumption) while keeping the timer module active to wake up the MCU after a period of time. To do so, Timer/Counter0 should be configured in *asynchronous mode* that allows it to use an external 32.768kHz crystal (`TOSC1`) to operate independently from the main oscillator of the MCU.

However, the asynchronous mode of Timer/Counter0 requires a number of safety measures as listed in the datasheet of the MCU (Atmel (2011), pp. 106–108). We limit the description herein to two important ones:

- The first precaution that should be considered when operating in asynchronous mode is the stabilization of the timer after wakeup. Indeed, the datasheet stipulates that "*if the time between wakeup and re-entering*



(a) Stabilization of Timer/Counter0.



(b) TWI serial transfer using interrupts.

Figure 16: Examples of ADPs used in the experiments.

sleep mode is less than one `TOSC1` *cycle, the interrupt will not occur, and the device will fail to wake up*".

- Also, the datasheet indicates that "*when writing to one of the [timer] registers, the value is transferred to a temporary register, and latched after two positive edges on* `TOSC1`*. The user should not write a new value before the contents of the Temporary Register have been transferred to its destination*".

To ensure both requirements, the same mechanism is generally employed, which is based on polling the first three bits of the `ASSR` register that indicate the effective transfer from the temporary register to the actual register. Since this operation requires at least one `TOSC1` cycle, the timer driver can assess the value of these bits for ensuring both stabilization and correct transfer to registers.

Consequently, we wrote three ADPs to ensure that the driver performs the appropriate polling mechanism. The first one, denoted $\mathcal{A}_{\texttt{STBL}}$, specifies the stabilization requirement and verifies that, when the MCU is waked-up by the timer routine, at least one of the timer registers (`OCR0`, or `TCCR0`) is modified and the program will not return to sleep again only until it verifies that appropriate status bit in `ASSR` indicates the end of transfer. An illustration of this ADP is depicted in Fig. 16(a). The two other ADPs, denoted $\mathcal{A}_{\texttt{OCR0}}$ and $\mathcal{A}_{\texttt{TCCR0}}$, model the proper access to registers `OCR0` and `TCCR0` respectively, and ensure that the driver waits after every write access for the completion of the transfer operation before modifying the register again.

8.1.2. ADG715 Analog Switch

The component ADG715 is an analog switch that is used generally as a multiplexer to dynamically route the power supply to other components (mainly sensors) for controlling energy consumption. It is configured by the MCU through a TWI (*Two Wire Interface*) serial bus for sending byte commands to open/close the switch ports.

To ensure the proper transfer of these commands over the bus, several safety rules should be observed. In our experiments, we were interested in two properties:

- It is important to ensure that all TWI-related hardware registers are not be modified when the bus is busy. In the case of TinyOS (both 1.x and 2.x), TWI operations are performed in the interrupt-based mode. Therefore, the $\mathcal{A}_{\texttt{TWI-TX}}$ ADP, depicted in Fig. 16(b), tracks the start of a transmission and performs an asynchronous transition to model the arrival of the interrupt at any moment. All access to the register in the meanwhile are forbidden.

- In addition to safe serial transfer, the `SCL` and `SDL` pins, used as the clock and data lines respectively, should be configured as pulled-up. Our ADP $\mathcal{A}_{\texttt{PULL-UP}}$ verifies this condition by checking that the corresponding GPIO pins are configured as input pins through the `DDx` register and that they are driven high through the `PORTx` register. This check is performed at every transmission over the TWI bus.

8.1.3. CC2420 Transceiver

In Section 3, we described in detail the ADP $\mathcal{A}_{\texttt{SPI-TX}}$ that models a safe serial transfer between the MCU and an SPI slave (which is the CC2420 transceiver in the case of the MicaZ mote). In addition, our benchmark includes a second ADP $\mathcal{A}_{\texttt{SPI-SS}}$ that ensures that the MCU selects the appropriate end-point slave by pulling down a particular `SS` (*Slave Select*) pin before starting transmitting over the SPI bus. Consequently, the ADP checks that direction and state of PB0 pin are correctly set in registers `DDRB` and `PORTB` whenever a byte is written into the data register `SPDR`.

8.2. Results and Discussion

The obtained results are summarized in Table 1 and 2. For each ADP, we analyzed the original device driver along with a modified incorrect version that contains a manually injected bug. Note that two of our benchmark device drivers contained errors in their original version that we discovered during our experiments. We used a timeout of 30 minutes and we reported the total analysis time, the peak memory consumption and the analysis result. We distinguish between four types of results: (*i*) safe program with no missed errors (✔), (*ii*) the program is incorrect and the analysis reports no error (⚠), (*iii*) the program

is incorrect and the errors are detected (✘), and finally (*vi*) the program is correct while false alarms are reported (⚠). A precise analysis should only exhibit (*i*) and (*iii*) results. For certifications purposes, a small ratio of (*vi*) is expected due to approximations, but (*ii*) should never occur. For bug finding, however, (*ii*) is acceptable but not (*vi*).

8.2.1. TinyOS Benchmarks

Table 1 shows the results of our tool SADA on the different TinyOS device drivers. In order to show the cost of introducing arbitrary preemption during the analysis, we considered both sequential and preemptive execution models separately. In addition, we compared the results of the hardware state partitioning domain $\mathcal{D}_{\mathcal{S}}$ with the tasks queue partitioning domain $\mathcal{D}_{\mathcal{Q}}$. Overall, our benchmarks consisted of 112 tests and we analyzed a total of 23260 lines of code, with drivers containing between 1 to 10 tasks and 1 or 2 interrupts.

The analysis terminated before timeout in 95% of the test cases, with 90% of them under one minute. We experienced only 6 timeouts. Four of them were due to the coarse over-approximation of the state partitioning domain $\mathcal{D}_{\mathcal{S}}$ that does not keep information about the tasks queue, but these timeouts have been eliminated by the use of the more precise domain $\mathcal{D}_{\mathcal{Q}}$. The two remaining ones emerged when analyzing the incorrect versions of the drivers. Nevertheless, it is important to note that these benchmarks were run in full-coverage mode of SADA, *i.e.* the analysis does not stop until all paths are verified. SADA supports also another option that terminates the analysis whenever an error is detected. In such configuration, all timeouts disappear, as described later in this section (see Table 2).

In terms of precision, we note that SADA has not missed any bug on the incorrect version of the drivers, which is coherent with the soundness property of the underlying abstract interpretation theory. In addition, two real bugs were detected on the original versions, which were, to our knowledge, not known before. On the other hand, 7 false alarms were detected. Using the tasks queue partitioning domain $\mathcal{D}_{\mathcal{Q}}$ we can eliminate 4 of them, in a similar way to the example in Fig. 11 that motivated the introduction of $\mathcal{D}_{\mathcal{Q}}$. The remaining 3 false alarms are all related to the timer driver and are due to the lack of quantitative modeling of the physical time and delays. Indeed, asynchronous events – such as propagating a value to the `TCCR0` from the temporary register – are modeled in our analysis using nondeterminism and can occur at any moment. However, these transitions in fact have a limited trigger timeframe, after which we are sure that the asynchronous event has occurred. This type of information is not handled by SADA and is out of the scope of the ADPs semantics. Note that other existing analysis tools such as i-CBMC do not model such semantics and are therefore prone to the same problem.

From these results, we can observe that the analysis costs do not always increase with the level of precision of the abstract domain. Indeed, in most cases, the queue partitioning domain $\mathcal{D}_{\mathcal{Q}}$ has shown a better analysis time compared to the more compact state abstraction $\mathcal{D}_{\mathcal{S}}$. This is particularly observable when the driver implementation uses a significant number of tasks, such as the CC2420 2.x driver where we obtained a 95% decrease in the analysis time. This is explained by the fact that, due to the additional details provided by the increased precision of the domain, more spurious execution paths are filtered and fewer iterations are required to reach the fixpoint. However, for less task-intensive programs such the Timer driver, the domain $\mathcal{D}_{\mathcal{Q}}$ was less efficient.

Finally, these experimental results demonstrate the scalability of the Modular Abstract Interpretation framework, since the sound preemptive analysis was able to analyze the full search space with arbitrary preemption, while maintaining a reasonable cost in comparison to the restricted sequential analysis which does not necessarily cover all behaviors.

8.2.2. Comparison with i-CBMC

Table 2 shows the obtained results of running the TinyOS 2.x test cases using i-CBMC with different loops unwinding. The table also presents the results of SADA without full-coverage in case of error detection, since i-CBMC behaves in the same way. To compare both approaches, we consider three criterions: efficiency, precision and automation.

Efficiency It is clear that SADA scales better than i-CBMC in all test cases. Analysis times of i-CBMC are larger in general with a total of 10 timeouts, while SADA converged in all cases without exceeding one minute per driver. Note that we limited the maximal unwinding of the main TinyOS loop to two iterations in the experiments with i-CBMC, since the analysis time of i-CBMC exceeded the timeout duration when using three loops unwinding for all the benchmark programs. In addition, the memory consumption of i-CBMC is much higher, reaching the giga byte in many cases, in contrast to SADA that consumed at most 20 mega bytes in the worst case thanks to the use of the efficient abstraction of the interval domain.

Precision Due to the limited search depth in i-CBMC, not all errors can be discovered. This is exemplified by the missed bugs reported during the analysis of the incorrect versions of the drivers. SADA, and abstract interpretation based tools in general, do not suffer from these limitations since all possible errors are detected, which makes the tool more adequate to correctness certification. False alarms are, on the other hand, the main drawback of our method since no indication can be provided to developers to decide the genuineness of the errors. However, in practice, SADA presents a low false alarm rate with only one

Table 1: Analysis benchmarks for TinyOS 1.x and 2.x. Three metrics are shown: the analysis time (in seconds), the peak memory consumption (in mega bytes) and the analysis result (✔: safe, ✘: bug detected correctly, ⚠: false bug alarm, ⚠: bug missed). ∞ denotes a timeout of 30mn.

						Non-Preemptive Analysis								Preemptive Analysis							
						$\mathcal{D}_S$				$\mathcal{D}_Q$				$\mathcal{D}_S$				$\mathcal{D}_Q$			
Driver	# Lines	# ISR	# Tasks	ADP	# States	Original			Incorrect			Original			Incorrect			Original			Incorrect		
Timer 1.x	1627	1	3	$\mathcal{A}_{\texttt{STBL}}$	7	1	10	⚠	1	10	✘	1	10	✔	1	10	✘	6	11	⚠	7	11	✘	39	16	⚠	36	16	✘
				$\mathcal{A}_{\texttt{OCRO}}$	4	1	10	⚠	1	10	✘	1	10	✔	2	10	✘	3	10	⚠	3	10	✘	29	15	⚠	29	15	✘
				$\mathcal{A}_{\texttt{TCCRO}}$	4	1	10	✔	1	10	✘	1	10	✔	1	10	✘	3	10	✔	3	10	✘	37	16	✔	41	15	✘
Timer 2.x	2384	2	2	$\mathcal{A}_{\texttt{STBL}}$	7	2	11	✔	3	11	✘	2	11	✔	3	11	✘	7	12	✔	15	12	✘	26	13	✔	78	15	✘
				$\mathcal{A}_{\texttt{OCRO}}$	4	3	11	✔	2	11	✘	4	11	✔	3	11	✘	10	12	✔	9	12	✘	38	13	✔	36	13	✘
				$\mathcal{A}_{\texttt{TCCRO}}$	4	3	11	✔	3	11	✘	3	11	✔	3	11	✘	10	11	⚠	10	12	✘	23	13	⚠	23	13	✘
ADG715 1.x	2038	1	1	$\mathcal{A}_{\texttt{PULL-UP}}$	4	1	11	✘	1	11	✘	1	11	✘	1	10	✘	1	11	✘	1	11	✘	1	11	✘	1	11	✘
				$\mathcal{A}_{\texttt{TWI-TX}}$	6	1	11	✔	1	11	✘	1	11	✔	2	11	✘	4	11	✔	4	11	✘	7	12	✔	7	12	✘
ADG715 2.x	4412	1	6	$\mathcal{A}_{\texttt{PULL-UP}}$	4	3	13	✔	3	13	✘	2	13	✔	2	13	✘	23	14	✔	30	14	✘	8	13	✔	6	14	✘
				$\mathcal{A}_{\texttt{TWI-TX}}$	6	4	14	✘	6	14	✘	2	14	✘	3	14	✘	40	16	✘	42	16	✘	6	14	✘	6	14	✘
CC2420 1.x	2666	1	1	$\mathcal{A}_{\texttt{SPI-SS}}$	4	10	13	✔	4	13	✘	28	12	✔	2	12	✘	12	12	✔	6	12	✘	850	42	✔	52	11	✘
				$\mathcal{A}_{\texttt{SPI-TX}}$	10	5	12	✔	3	12	✘	28	13	✔	26	12	✘	21	13	✔	19	14	✘	1600	74	✔		∞	
CC2420 2.x	10133	2	10	$\mathcal{A}_{\texttt{SPI-SS}}$	4	1040	33	⚠	800	33	✘	12	17	✔	5	16	✘		∞			∞		34	18	✔	27	18	✘
				$\mathcal{A}_{\texttt{SPI-TX}}$	10	1659	27	⚠	597	27	✘	13	17	✔		∞			∞			∞		39	18	✔	49	19	✘

Table 2: Comparison with i-CBMC on the TinyOS 2.x drivers with different unwinding iteration limits.

	Non-Preemptive Analysis												Preemptive Analysis											
	i-CBMC (1 iteration)						i-CBMC (2 iterations)						SADA						i-CBMC (1 iteration)						i-CBMC (2 iterations)						SADA					
ADP	Original			Incorrect			Original			Incorrect			Original			Incorrect			Original			Incorrect			Original			Incorrect			Original			Incorrect		
$\mathcal{A}_{\texttt{STBL}}$	38	200	✔	29	200	⚠	456	447	✔	354	441	✘	2	11	✔	1	9	✘	45	290	✔	43	286	⚠	404	494	⚠	386	497	✘	26	13	✔	6	12	✘
$\mathcal{A}_{\texttt{OCRO}}$	16	178	✔	35	247	⚠	325	450	✔	401	474	✘	4	11	✔	1	8	✘	33	262	✔	64	340	⚠	354	500	✔	444	561	✘	38	13	✔	1	9	✘
$\mathcal{A}_{\texttt{TCCRO}}$	29	210	✔	34	225	⚠	357	462	✔	400	484	✘	3	11	✔	1	10	✘	54	323	⚠	56	336	✘	385	510	⚠	397	507	✘	4	10	⚠	3	10	✘
$\mathcal{A}_{\texttt{PULL-UP}}$	2	37	✔	1	37	⚠	1601	1919	✔	1552	1907	⚠	2	13	✔	1	11	✘	1	37	✔	1	41	✘	1	36	✔	1	37	⚠	8	13	✔	4	12	✘
$\mathcal{A}_{\texttt{TWI-TX}}$	1	37	⚠	1	36	⚠	1362	1762	⚠	1520	1940	⚠	1	10	✘	1	11	✘	1	36	⚠	1	37	⚠	1	36	⚠	1	37	⚠	1	11	✘	1	11	✘
$\mathcal{A}_{\texttt{SPI-SS}}$	2	64	✔	1	62	⚠	129	507	✔	126	505	⚠	12	17	✔	4	16	✘		∞			∞			∞			∞		34	18	✔	6	16	✘
$\mathcal{A}_{\texttt{SPI-TX}}$	2	65	✔	2	62	⚠		∞			∞		13	17	✔	1	14	✘		∞			∞			∞			∞		39	18	✔	2	15	✘

occurence in TinyOS 2.x benchmark. Note that i-CBMC reported also the same false alarm because both tools lack appropriate modeling of hardware-level timings in order to restrict the firing timeframes of the critical interrupts.

Automation As reported by Bucur and Kwiatkowska (2011), employing a bounded model checking approach is generally hampered by the laborious task of setting the appropriate loops unwinding which requires enumerating all loops of the program, eliminating the unnecessary ones and fixing an individual unwinding limit for the remaining loops (an exception was made for the main TinyOS loop for which we made its unwinding limit as a parameter of the analysis to vary the depth of execution flows). In the case of SADA, such manual tuning was not required thanks to the widening mechanism of abstract interpretation that allows a fully automatic and sound analysis up to arbitrary (possibly infinite) loop bounds. Additionally, i-CBMC is a general purpose tool for analyzing preemptive ANSI C programs and does not embed a dedicated semantics for low-level hardware interactions. As a consequence, it is necessary to manually modify the programs in order to emulate the reaction of the device to registers modifications. Practically, before analyzing a program with i-CBMC, we added C functions modeling the behavior of hardware in reaction to read/write register events and we inserted asynchronous calls to these functions and to interrupt vectors at the appropriate locations. Note that we were not able to faithfully mimic the full semantics of the driver due to some limitations in i-CBMC related to the management of atomic sections. Consequently, the obtained instrumented program is not semantically equivalent to the real behavior of the driver and the device.

9. Related Work

Software reliability in wireless sensor networks represents a crucial problem that has been addressed by many recent works. Due to the undecidability of the verification problem, proposed solutions tend to be limited to specific families of properties. We can distinguish between two major trends.

Network-level verification approaches focus on the distributed behavior of the sensor network. By allowing the users to specify inter-node assertions, these tools aim at checking the correctness of the message exchange protocols and the dynamics of the global state of the network.

T-Check (Li and Regehr (2010)) and KleeNet (Sasnauskas et al. (2010)) are considered as the pioneering network-level verification tools for wireless sensor networks. T-Check is a depth-bounded model checker that can verify safety and liveness properties expressed over the variables of the nodes of the entire network. To alleviate the problem of state space explosion, T-Check uses random walks at specific stages of the verification process and embeds a partial order reduction technique to avoid exploring redundant paths. KleeNet is based on the symbolic virtual machine KLEE (Cadar et al. (2008)) and aims at finding node interaction bugs by injecting specific failures (such as packet loss and node crash) in a nondeterministic way during the symbolic execution of the program.

Anquiro (Mottola et al. (2010)) is a domain-specific extension to the Bogor model checker (Robby et al. (2003)) that adds support for the specificities of sensor network programs written for the Contiki OS (Dunkels et al. (2004)), such as timers and wireless message processing. Basically, the main idea of Anquiro is to translate the semantics of the Coniki program into a finite state machine that can be processed by the Bogor model checker. The user can express a LTL formulae specifying a property about the program's variables and that can be quantified over the network nodes to model correctness rules of communication protocols.

In contrast to the network-level approaches, *node-level* verification approaches are rather interested in the local behaviors of individual nodes, and are therefore more adapted to device drivers verification. Indeed, the previously described category slices the programs at a high level of abstraction in order to effectively handle the complexity of networks interactions. Consequently, access to low-level hardware details, such as bitwise operations on registers, are not taken into consideration.

The most widespread program verification technique is *testing* that consists in the assessment of a limited number of finite program execution traces in order to look for the presence of some predefined errors. To perform such type of verification, a controlled runtime environment is required, that should be able to monitor the correct evolution of the program state during execution. Generally, this is done by instrumenting the program with a runtime detection mechanism of error situations that puts the program into a safe mode and alerts the user whenever an error is detected. Several solutions has been proposed in this context that are tailored to TinyOS programs (Bucur (2012); Zhai et al. (2014)) and they allow tracking complex safety properties during execution on real hardware platforms. RID (Regehr (2005)) is another testing tool for TinyOS which is, in contrast to previously cited work, particularly designed to run in an emulation environment. It is based on a random testing technique that employs a *restricted interrupt discipline* which guides the scheduling of random interrupts in order to fire them at semantically valid moments and avoid spurious executions.

Testing techniques are well-appropriate to bug finding but lack a formal framework that allows assessing properties on a larger scale than individual executions. To this end, bounded model checking (Clarke et al. (2001)) has been proposed to exhaustively analyze a part of the search space using a symbolic encoding of execution traces truncated to a given length. Two verification tools have imple-

mented this technique for TinyOS and both are based on CBMC (Clarke et al. (2004)), a general purpose ANSI C model checker. The first tool is TOS2CProver (Bucur and Kwiatkowska (2011)) and provides a verification toolchain of TinyOS programs for MSP430 MCU. Before applying CBMC, it instruments the source with appropriate assertions expressing generic language safety rules or particular requirements on hardware registers values. Since CBMC can handle only sequential programs, TOS2CProver applies the sequentialization technique and inserts nondeterministic calls to interrupt handlers at specific locations using a partial order reduction, in order to decrease the program's state space. The second tool is i-CBMC (Kroening et al. (2015)), which is an extension of CBMC to support native modeling of interrupt preemption without applying any partial order reduction. The basic idea of i-CBMC is to enrich the trace formula of CBMC with an encoding of the possible interrupts interleavings using a set of symbolic clocks. These clocks represent the logical occurrence time of the access events on the program variables. The proposed method defines a set of constraints for restricting the values of theses clocks and expressing a set of happens-before conditions emerging from the semantics of interrupt preemption.

Finally, sound formal verification has also been proposed for interrupt-based programs. Brauer et al. (2010) developed an abstract interpreter for analyzing binary programs of the ATmega16 microcontroller. Their solution is tailored to the static verification of generic language errors, such as out-of-bound array access. They proposed to use a reduced product of word-level and bit-level intervals in order to handle both arithmetic and bitwise operations, the latter being omnipresent in binary codes. The analysis of interrupts is context-sensitive and employs only the information of the global interrupt bit to restrict the occurrences of interrupts. However, their approach does not consider the presence of nested interrupts nor the asynchronous concurrency of hardware operations.

10. Conclusion

We presented an effective static analysis by Abstract Interpretation of device drivers in TinyOS programs. We described an automata-based formalism to express functional properties that specify the correct hardware interaction patterns that should be followed by the device driver. Our analysis is based on several abstract domains that provide a multi-level partitioning according to the hardware state, the tasks queue and the interrupts masking system. To efficiently handle concurrency, we perform a compositional analysis by processing the interrupt vectors separately and propagate their effects to program locations where interrupts are enabled. Several experiments were conducted on real-world TinyOS drivers and promising results demonstrate the efficiency of the approach.

We plan to extend the presented analysis to other execution models of sensor network programs. An interesting case is the *protothreads* paradigm (Dunkels et al. (2006)) implemented in the Contiki operating system. This programming abstraction is different from the task-driven execution model of TinyOS and allows developing programs in a thread-like style, which is more "natural" in the context of event-driven programs and has been proved to reduce their implementation complexity.

In addition, our current implementation supports only the ATmega128 MCU and we would like to extend our framework to other microcontroller families. We envisage to add support for the famous MSP430 16-bits MCU, and also the promising ARM Cortex M0 32-bits architecture implemented in various MCUs, such as Nordic nRF51 and STM32 F0 MCUs. To support these platforms, new ADPs should be formalized to express the specific hardware behaviors of their different subsystems.

References

Atmel, 2011. Atmega128(l) datasheet. www.atmel.com/images/doc2467.pdf.

Atzori, L., Iera, A., Morabito, G., 2010. The internet of things: A survey. Computer Networks 54, 2787 – 2805.

Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X., 2002. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, in: The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones. Springer. volume 2566 of *Lecture Notes in Computer Science (LNCS)*, pp. 85–108.

Brauer, J., Noll, T., Schlich, B., 2010. Interval analysis of microcontroller code using abstract interpretation of hardware and software, in: Proc. of the 13th International Workshop on Software & Compilers for Embedded Systems (SCOPES), pp. 3:1–3:10.

Bucur, D., 2012. Temporal monitors for tinyos., in: Proc. of the Third International Conference on Runtime Verification (RV), Springer. pp. 96–109.

Bucur, D., Kwiatkowska, M., 2011. On software verification for sensor nodes. Journal of Systems and Software 84, 1693–1707.

Cadar, C., Dunbar, D., Engler, D., 2008. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs, in: Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI), pp. 209–224.

Clarke, E., Biere, A., Raimi, R., Zhu, Y., 2001. Bounded model checking using satisfiability solving. Form. Methods Syst. Des. 19, 7–34.

Clarke, E., Kroening, D., Lerda, F., 2004. A tool for checking ANSI-C programs, in: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Springer Berlin Heidelberg. volume 2988 of *Lecture Notes in Computer Science*, pp. 168–176.

Cousot, P., Cousot, R., 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), ACM. pp. 238–252.

Cousot, P., Cousot, R., 1992. Abstract interpretation frameworks. Journal of Logic and Computation 2, 511–547.

Cousot, P., Cousot, R., 2002. Modular static program analysis, invited paper, in: Proc. of the Eleventh International Conference on Compiler Construction (CC), Springer. pp. 159–178.

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X., 2005. The Astrée analyzer, in: Proc. of the European Symposium on Programming (ESOP), Springer. pp. 21–30.

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X., 2009. Why does astrée scale up? Formal Methods in System Design 35, 229–264.

Cousot, P., Halbwachs, N., 1978. Automatic discovery of linear restraints among variables of a program, in: Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), ACM. pp. 84–97.

Dunkels, A., Gronvall, B., Voigt, T., 2004. Contiki - a lightweight and flexible operating system for tiny networked sensors, in: Proc. of the 29th IEEE International Conference on Local Computer Networks (LCN), pp. 455–462.

Dunkels, A., Schmidt, O., Voigt, T., Ali, M., 2006. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems, in: Proc. of the 4th International Conference on Embedded Networked Sensor Systems (SensSys), ACM. pp. 29–42.

Feret, J., 2001. Occurrence counting analysis for the pi-calculus. Electronic Notes in Theoretical Computer Science 39.(2). Workshop on GEometry and Topology in COncurrency theory.

Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D., 2003. The nesC language: A holistic approach to networked embedded systems, in: Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI), ACM. pp. 1–11.

Jeannet, B., Miné, A., 2009. Apron: A library of numerical abstract domains for static analysis, in: Proc. of the 21st International Conference on Computer Aided Verification (CAV), Springer-Verlag. pp. 661–667.

Kaminski, M., Francez, N., 1994. Finite-memory automata. Theoretical Computer Science 134, 329 – 363.

Kroening, D., Liang, L., Melham, T., Schrammel, P., Tautschnig, M., 2015. Effective verification of low-level software with nested interrupts, in: Proc. of the Design, Automation & Test in Europe Conference & Exhibition (DATE), EDA Consortium. pp. 229–234.

Levis, P., Madden, S., Polastre, J., Szewczyk, R., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., Culler, D., 2004. Tinyos: An operating system for sensor networks, in: Ambient Intelligence, Springer Verlag. pp. 115–148.

Li, P., Regehr, J., 2010. T-check: Bug finding for sensor networks, in: Proc. of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), ACM. pp. 174–185.

Miné, A., 2006a. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics, in: Proc. of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), ACM. pp. 54–63.

Miné, A., 2006b. The octagon abstract domain. Higher-Order and Symbolic Computation (HOSC) 19, 31–100.

Miné, A., 2011. Static analysis of run-time errors in embedded critical parallel C programs, in: Proc. of the 20th European Symposium on Programming (ESOP), Springer. pp. 398–418.

Miné, A., 2012. Abstract domains for bit-level machine integer and floating-point operations, in: Proc. of the 4th International Workshop on Invariant Generation (WING), p. 16.

Miné, A., 2014. Relational thread-modular static value analysis by abstract interpretation, in: Proc. of the 15th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), Springer. pp. 39–58.

Monniaux, D., 2007. Verification of device drivers and intelligent controllers: A case study, in: Proc. of the 7th ACM/IEEE International Conference on Embedded Software (EMSOFT), pp. 30–36.

Mottola, L., Voigt, T., Österlind, F., Eriksson, J., Baresi, L., Ghezzi, C., 2010. Anquiro: Enabling efficient static verification of sensor network software, in: Proc. of the Workshop on Software Engineering for Sensor Network Applications (SESENA), ACM. pp. 32–37.

Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W., 2002. CIL: Intermediate language and tools for analysis and transformation of C programs, in: Proc. of the 11th International Conference on Compiler Construction (CC), pp. 213–228.

Ouadjaout, A., Lasla, N., Bagaa, M., Badache, N., 2014. Poster abstract: Static analysis of device drivers in tinyos, in: Proc. of the 13th International Symposium on Information Processing in Sensor Networks (IPSN), IEEE Press. pp. 297–298.

Parikh, R.J., 1966. On context-free languages. Journal of ACM 13, 570–581.

Regehr, J., 2005. Random testing of interrupt-driven software, in: Proc. of the 5th ACM International Conference on Embedded Software (EMSOFT), ACM. pp. 290–298.

Robby, Dwyer, M.B., Hatcliff, J., 2003. Bogor: An extensible and highly-modular software model checking framework, in: Proc. of the 9th European Software Engineering Conference (ESEC/FSE), pp. 267–276.

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

Zhai, J., Sridhar, N., Hallstrom, J.O., 2014. Supporting the specification and runtime validation of asynchronous calling patterns in reactive systems, in: Proc. of the 5th International Conference on Runtime Verification (RV), Springer. pp. 108–123.

Abdelraouf Ouadjaout received the MS degree in computing science from the USTHB (Algiers, Algeria) in 2009 and is working toward the PhD degree in the same university. He was also a research assistant at the Research Center CERIST (Algiers, Algeria) from 2009 to 2015. He is currently a research engineer at ENS and UPMC (Paris, France). His current research interests include wireless sensor networks and software verification of embedded software.

Antoine Miné is a Computer Science Professor at Universit Pierre et Marie Curie (Paris, France). After a PhD at cole Polytechnique in 2004, he was Research Scientist at CNRS (2007-2015) at cole normale suprieure. He specializes in formal methods ensuring program correctness, and in particular the theory of Abstract Interpretation and its application to design semantic-based automatic static analyzers. His contributions include the design of novel abstractions to prove numerical properties of programs, and the participation to the design, development and industrialization of the Astrée analyzer that proved the correctness of large embedded critical avionics C codes.

Noureddine Lasla received the MS degree in computing science from the National Institute of Informatics (ESI) in 2008 and is working toward the PhD degree at the USTHB, Algeria. He is also a research assistant at the Research Center CERIST, Algeria. His current research interest include localization, deployment and coverage in wireless sensor network.

Nadjib Badache joined USTHB University of Algiers, in 1983, as assistant professor and then professor, where he taught operating systems design, distributed systems and networking with research mainly in distributed algorithms and mobile systems. From 2000 to 2008, he was the head of LSI laboratory, where he conducted many projects on routing protocols, energy efficiency and security in mobile ad-hoc networks and WSN. Since March 2008, he has been the director of CERIST and professor at USTHB University.