Bounded Model Checking of Contiki Applications

Thilo Vörtler, Steffen Rülke
Fraunhofer Institute for Integrated Circuits IIS
Design Automation Division EAS
Zeunerstraße 38, 01069 Dresden, Germany
Email: {thilo.voertler|steffen.ruelke}@eas.iis.fraunhofer.de

Petra Hofstedt
Brandenburg University of Technology Cottbus
Department of Computer Science
03013 Cottbus, Germany
Email: hofstedt@informatik.tu-cottbus.de

***Abstract*—Verification of embedded systems is a challenge due to the tight combination of hardware and software. We present an approach on the automatic verification of embedded system applications for the operating system Contiki using a standard bounded model checking tool for software. By using an operating system a higher abstraction level to hardware is possible. Our approach is therefore easily applicable for the verification of different hardware platforms.**

I. Introduction

The number of embedded systems (ESs) we are faced with e.g. in consumer electronics, cars, and medical devices is growing more and more. ES hardware is usually composed of standard components. Due to hardware (HW) limitations and low power constraints designing software (SW) for ESs can be tedious. The code has to be highly optimized and is often interrupt driven. Furthermore for SW development also the environment of the system must be considered. As ESs are often used in safety critical applications it is even more important to avoid SW bugs. One trend in recent years has been the use of event driven operating systems like Contiki [1], [2] or TinyOS [3] that run on memory constrained low power systems consisting only of several kilobytes of RAM.

The approach described in this paper allows it to find SW bugs by static analysis in applications written for the operating system Contiki before the ES is deployed. Our modelling approach for formal verification accomplishes a high abstraction layer as it models the ES at the driver level between operating system and HW. Therefore, it is easily portable to model different HW platforms. As formal verification technique we use bounded model checking (BMC) [4] and the open source model checking (MC) tool CBMC [5] in this work.

To check ESs already at compile time *static code analysis* can be applied. In [6] a survey of related SW verification techniques is presented. The formal verification of embedded system SW requires suitable modelling approaches. Such an approach is described in [7], code is split up into HW-dependent and HW-independent parts. Several MC tools (including CBMC used in our work) are discussed. However the use of an embedded operating system is not considered.

A complete formal verification flow for TinyOS using CBMC is shown in [8]. As TinyOS applications are written in nesC the code has to be translated into C to be used as input for the MC tool. However, the abstraction level of the models is low as it is done on the level of registers and ports of the target CPU. In [8] this is done for an MSP430 micro-controller [9].

In [10] the MC tool Anquiro is presented. Anquiro supports the formal verification of Contiki applications. However, Anquiro focusses on the verification of network applications running on several nodes without any detailed access to external HW and handling of interrupts. As Anquiro is based on the Bogor Model Checker it requires code transformation.

As far as we know our work presents the first formal verification approach for Contiki-based applications including interrupts. Working on a high abstraction layer makes it unnecessary to model particular HW registers. Furthermore, it is possible to fully automate loop unwinding for BMC in our approach. We use unmodified applications without any intermediate translations as input for MC. Therefore, generated counterexample traces can be debugged easily.

II. Preliminaries

1) BMC of Software: For our work CBMC was used as MC tool. CBMC is a BMC based tool for the verification of software written in ANSI-C, and is therefore suitable for the verification of ES code. In BMC state space search is limited by an upper bound and the MC problem is transformed into a satisfiability problem. For verification of SW using BMC this means, that an upper bound for all loops in a program has to be defined, up to which the system is unwound.

In CBMC the properties to be checked for a program extend the source code. The command `assert(x<5)` causes CBMC to check for all program executions, whether it is possible to violate this property, that means whether `x` may take the value 5 or larger. Furthermore, CBMC can automatically generate properties, including checks for correct array bounds, numerical overflow, and null pointer dereference. CBMC also generates unwinding assertions, which check that no program execution path exists that has been cut off by limiting the loop unwinding. To model non-deterministic behaviour caused by the system environment e.g. by user input to the system, non-determined variables are used. For example in:

```
int temp = nondet_int();
__CPROVER_assume(temp>=0 and temp <=10);
```

the variable `temp` is declared as a non determined variable, restricted to all possible integer values in the interval 0..10.

2) Operating System Contiki: The main focus of Contiki are low powered and memory constrained ESs. It uses therefore an event driven kernel. Applications for Contiki

```
PROCESS_THREAD(blink_process, ev, data) {
 PROCESS_BEGIN();
 while(1) {
   static struct etimer et;
   etimer_set(&et, CLOCK_SECOND);
   PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
   leds_on(LEDS_ALL);
   etimer_set(&et, CLOCK_SECOND);
   PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
   leds_off(LEDS_ALL); }
PROCESS_END(); }
```

Fig. 1. Contiki example application using Protothreads

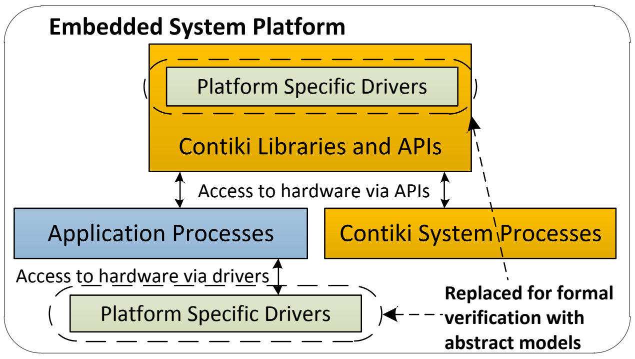

Embedded System Platform
Platform Specific Drivers
Contiki Libraries and APIs
Access to hardware via APIs
Application Processes
Contiki System Processes
Access to hardware via drivers
Platform Specific Drivers
Replaced for formal verification with abstract models

Fig. 2. Replacement of drivers for formal verification

are written using standard C with added macros to simplify programming (*protothreads* programming model [11]). The event driven kernel of Contiki consists of events and processes, where events are used to invoke processes of the system. The events are stored in an event queue. Processes in Contiki are applications or system tasks, and only one process is actively running on the processor of the system at a time. As no pre-emption mechanism exists tasks have to hand over control to other threads. The main task of the Contiki scheduler (or main loop) is, to take an event from the event queue and invoke the related process. In Fig. 1 an example of a Contiki application (part of the official Contiki release) using *protothreads* is shown. The application periodically blinks LEDs of the system. The macro `PROCESS_WAIT_EVENT_UNTIL` is used to suspend the process until a timer is expired, which is defined by the event timer function `etimer_set`.

Contiki is designed as a portable operating system for different HW platforms, therefore, for many tasks libraries exist, which abstract access to HW specific drivers. The event timer system used in the example in Fig. 1 allows it to suspend a process for a specific time period. After the time is over the waiting process is reactivated using an event. The HW realization of the timing system is completely encapsulated from the application. Additionally, application programming interfaces (API) provide access to often used devices in ESs. The LED API functions `led_on` and `led_off` used in Fig. 1 belong to the Contiki system, and can be provided by each HW platform. If no appropriate API exists additional platform specific drivers must be written. These drivers then access the HW directly using e.g. ports of the processor.

III. Modelling a System for Verification

This section describes how an ES, which uses the operating system Contiki, is modelled for formal verification using CBMC. The input for the model checker shall be a modified version of the Contiki source code including HW models describing the environment of the system, the application to be verified, as well as annotated properties. Using these properties it is checked whether the SW accesses the HW correctly, uses the operating system correctly, and whether it is algorithmically correct. Therefore, three kinds of properties are considered:

- *Application dependent properties* check whether the application performs algorithmically correct.
- *Platform dependent properties* are used to check, whether processes correctly access the drivers of a HW component. These properties have to be defined once for each embedded system platform and must hold for all applications running on that platform.
- *Contiki specific properties* have to be valid for all Contiki applications independent of a specific embedded system platform. They check e.g. that an API is accessed correctly. As these properties are HW independent they have to be written only once and must hold for all Contiki applications.

The *application dependent properties* are optional.

To verify the entire ES consisting of the Contiki operating system and the application, all accesses to the HW have to be modelled. This is done by replacing the original drivers by abstract models with added properties, which check their correct usage. The challenge in writing these models is to describe all environment behaviours without generating unrealistic counter examples. In this case the driver model has to be refined. Furthermore, by modelling the HW access at the level of a driver interface the actual implementation of the driver is not verified. Consequently no bugs can be detected, which are caused by the driver implementation.

Fig. 2 shows which parts of the embedded system SW have been replaced in our approach for the formal verification. By replacing the HW drivers the implementation of the event system of Contiki is also checked. We found that replacing this part is unnecessary, as it is very small and allows to find bugs in the implementation as well as the configuration of Contiki (e.g. too small event queue size).

A. Replacing drivers

We divide HW drivers into two groups:

1) Hardware drivers that do not use interrupts: An example of a driver abstraction of this group is shown in Fig. 3. In Fig. 3a and 3b a function part of a memory card driver and the corresponding verification model are shown. Depending on, whether a memory card is available or not, either `MMC_SUCCESS` or `MMC_INIT_ERROR` are the return values of the function. Therefore, these values are also the return values of the abstract model. A driver function that does not return values is the LCD driver shown in Fig. 3c and Fig. 3d. In this case the abstract model checks, whether the HW is accessed correctly by passing the correct function parameters. It is checked that the display was correctly initialized, whether

```
char mmc_ping(void) {
if (!(MMC_CD_PxIN & MMC_CD))
  return (MMC_SUCCESS);
else
  return (MMC_INIT_ERROR);}
```

(a) Cut-out from memory card driver

```
char mmc_ping(void) {
char rvalue = nondet_char();
__CPROVER_assume(rvalue == MMC_SUCCESS
|| rvalue == MMC_INIT_ERROR);
return rvalue; }
```

(b) Memory card model for verification

```
void lcd_disp_char(uint8_t pos, uint8_t index) {
LCDMEM[pos+LCD_MEM_OFFSET] &= ~LCD_Char_Map[8];
if(pos < LCD_NUM_DIGITS ) {
 if(index < LCD_MAX_CHARS ) {
  LCDMEM[pos+LCD_MEM_OFFSET]|=LCD_Char_Map[index];
} } }
```

(c) Cut-out from LCD driver

```
void lcd_disp_char(uint8_t pos, uint8_t index) {
assert (init == 1);
assert(pos < LCD_NUM_DIGITS);
assert(index < LCD_MAX_CHARS);}
```

(d) LCD model for verification

Fig. 3. Replacement of hardware drivers without interrupts

```
while(1) {
  // Run a process from the event queue
  process_run();
  // Call the interrupt routine
  clock_interrupt();
  // Check whether system time has changed
  etimer_request_poll(); }
```

(a) Main loop of the verification platform using interrupts

```
void clock_interrupt(void) {
  long interval = nondet_long();
  __CPROVER_assume(interval >= 0);
  count=count+interval;
  seconds = seconds+(interval/CLOCK_CONF_SECOND);}
```

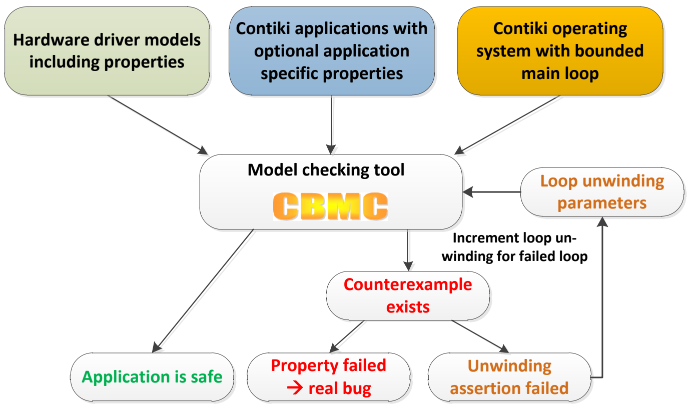
(b) Implementation of the clock system interrupt function

Fig. 4. Implementation of interrupts for our verification platform

the display position of the symbol to be displayed is valid, and whether the symbol is a supported symbol of the LCD.

2) Hardware drivers relying on interrupts: In low power ESs interrupts are often used to avoid busy waiting, when an external device is polled for a certain value. An interrupt can occur at any time during the program execution and introduces a kind of parallelism into the system. To model this parallelism [8], [12] propose to insert a call to an interrupt handling function after each C statement. We do not implement one of these approaches. In our case, interrupt routines are called during the main loop between the execution of processes. This means errors which are caused by the occurrence of interrupts during the execution of a process are not detected.

One part of Contiki that is often implemented using interrupts is the event timer system, which allows applications to sleep for a certain time period. A watchdog timer wakes up the system using an interrupt after a specified time has passed. Our approach to handle interrupts is shown in Fig. 4a. In the main loop of the our Contiki system the



Hardware driver models including properties
Contiki applications with optional application specific properties
Contiki operating system with bounded main loop
Model checking tool
CBMC
Loop unwinding parameters
Increment loop unwinding for failed loop
Counterexample exists
Application is safe
Property failed → real bug
Unwinding assertion failed

Fig. 5. Verification flow: Model checker input and possible results

function `clock_interrupt` is called, which implements the incrementation of the system time, done in HW using an interrupt routine. Afterwards `etimer_request_poll` is called to inform the event timer system of Contiki that a time period could have changed. In Fig. 4b the implementation of the interrupt routine is shown. The variables `count` and `seconds` are used to store the system time. Using these variables the event timer system checks whether a certain time has passed. Each time the interrupt routine is called the system time is incremented by an arbitrary value. This allows it to check applications for all possible time points.

B. Verification flow

The complete verification flow is shown in Fig. 5. The input into the MC tool is the Contiki application which should be verified, the abstract models describing the HW environment of the system, and the Contiki operating system implementation. In addition, the main loop of the Contiki system (see Fig. 4a) must be bound for BMC restricting the state space search to a finite number of events to be processed. All other loops, especially, in application processes must terminate as they would otherwise block the system because no pre-emption mechanism to suspend processes exists. Looking at the example application in Fig. 1 the loop from line 3 to 10 is bound, due to the `PROCESS_WAIT` macro.

As shown in Fig. 5 two results are possible when running the verification. When the verification succeeds no counterexample exists for the number of events checked. When a counterexample exists, this can either be an actual bug or the violation of an unwinding assertion. As the main loop was bound using a script, all loops can be unwound automatically to the needed depth by incrementing the loop unwinding bound for the failing loop. Unintentionally unbounded loops can be detected by manual inspection of loops, which have a high unwinding bound.

IV. Experiments and Results

We examined our approach using three Contiki applications, which are running on embedded system HW consisting of an MSP430 micro-controller, an acceleration sensor, LEDs, and

TABLE I
VERIFICATION RESULTS FOR EXAMPLE APPLICATIONS[1]

	Blink LED[2]	**Bubble sort n = 2**[2]		**Bubble sort n = 5**[2]		**Fall detection**[3]	
properties	generated	custom	generated	custom	generated	custom	generated[4]
number of program loops	25	31	31	31	31	55	55
number of verified properties	231	23	271	23	271	5	43
main loop unwindings	4	3	3	3	3	4	4
clauses	263819	1941310	2077568	4733167	5050188	247559213	248068798
variables	731999	652571	695348	1602805	1701370	85522070	85756339
verification time in seconds	11.4	12.6	17.5	48.6	53.9	5325	5521

[1] Tests run on a Intel Xeon X5675 with 3.06GHz, 48Gb RAM [2] CBMC 3.9 [3] CBMC 4.1 [4] only array bounds check

an LC-Display. For the ES devices drivers were replaced with abstract models. In addition, the event timer system of Contiki was described as in Fig 4b. An overview of the verification results is shown in Table I. The column *generated* indicates the automatically generated checks for array bounds, numerical overflows, division by zero, and the absence of null pointer dereferences in the code. Verification time is the runtime reported by CBMC for a run, where all loops are unwound so that no unwinding assertions fail. The number of clauses and variables are a measure for the size of the SAT problem. A description of the verified applications and the checked properties follows:

1) Blink LED: This is the example application shown in Fig 1, which periodically blinks all LEDs connected to the system. For this simple application, only automatically generated properties were checked.

2) Bubble sort: This application reads n numbers from a memory and sorts them in ascending order using the bubble sort algorithm. The sorted numbers are written out on an LC-Display. By increasing the parameter n the problem size can be increased. In addition to properties, which check the correct use of the LCD an application specific property was defined, which checks that the sorting of the numbers is correct.

3) Fall detection: This is the most complex of the evaluated applications. Purpose of this application is to detect, if the wearer of the ES has fallen down, e.g. for medical purposes. Therefore, an acceleration sensor is used, when a certain acceleration threshold was exceeded an LED of the system is enabled, to show that an eventual fall was detected. In this application the Contiki sensor API was used, which periodically collects data from sensors and sends events to application processes, when new data is available.

The *Blink LED* and *Bubble Sort* application could be successfully verified using CBMC version 3.9, however, the tool crashed on the *Fall detection* application with an internal error. Using CBMC version 4.1. this application could be verified, but only for array bounds checks due to a tool problem.

In all applications we could find no bugs related to the Contiki source code. When adding bugs to our applications e.g. trying to display a not supported symbol on the LCD, these bugs were found and counter examples were generated. In general, due to the use of BMC the absence of bugs can only be shown with respect to the number of main loop unwindings. The checking of additionally generated properties lead only to a small runtime increase.

V. CONCLUSIONS

In this paper a new approach to the verification of ESs was presented, which is easily portable to different HW platforms. We could show, that our approach is suitable to verify real world applications written for an MSP430 micro-controller platform. For future work we intend to implement a technique for more complex handling of interrupts as described in [12]. Furthermore we want to prove completeness of the verification with a technique like k-induction [13].

REFERENCES

[1] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," *29th Annual IEEE International Conference on Local Computer Networks*, pp. 455–462, 2004.

[2] "Contiki OS," http://www.contiki-os.org/, Mar. 2012.

[3] "TinyOS," http://www.tinyos.net/, Mar. 2012.

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 193–207, 1999.

[5] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, Kurt Jensen and Andreas Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.

[6] V. D'Silva, D. Kroening, and G. Weissenbacher, "A Survey of Automated Techniques for Formal Software Verification," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008.

[7] L. Cordeiro, B. Fischer, H. Chen, and J. Marques-Silva, "Semiformal verification of embedded software in medical devices considering stringent hardware constraints," in *International Conference on Embedded Software and Systems*, 2009, pp. 396 –403.

[8] D. Bucur and M. Kwiatkowska, "On software verification for sensor nodes," *Journal of Systems and Software*, vol. 84, no. 10, pp. 1693–1707, 2011.

[9] "MSP430 Low Power Microcontroller," http://www.ti.com/430brochure, Mar. 2012.

[10] L. Mottola, T. Voigt, F. Österlind, J. Eriksson, L. Baresi, and C. Ghezzi, "Anquiro: enabling efficient static verification of sensor network software," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*. ACM, 2010, pp. 32–37.

[11] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems," in *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems*, 2006.

[12] B. Schlich, T. Noll, J. Brauer, and L. Brutschy, "Reduction of interrupt handler executions for model checking embedded software," in *Hardware and Software: Verification and Testing*, ser. Lecture Notes in Computer Science, K. Namjoshi, A. Zeller, and A. Ziv, Eds., vol. 6405. Springer, 2011, pp. 5–20.

[13] M. Sheeran, S. Singh, and G. Stålmarck, "Checking Safety Properties Using Induction and a SAT-Solver," in *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*. Springer, 2000.