Verification of Software for Contiki-based Low-Power Embedded Systems Using Software Model Checking

Von der Fakultät für MINT - Mathematik, Informatik, Physik,
Elektro- und Informationstechnik
der Brandenburgischen Technischen Universität Cottbus-Senftenberg
zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften
(Dr.-Ing.)

genehmigte Dissertation
vorgelegt von

Diplomingenieur
Thilo Vörtler

geboren am 28. April 1984 in Zwenkau

Gutachter: Prof. Dr. rer. nat. habil. Petra Hofstedt
Gutachter: Prof. Dr. Christoph Grimm
Gutachter: Prof. Dr.-Ing. Heinrich Theodor Vierhaus
Tag der mündlichen Prüfung: 24. November 2017

Danksagung

Viele Kollegen und Freunde haben mich in den letzten Jahren bei der Erstellung dieser Dissertation unterstützt. Hiermit möchte ich mich insbesondere bei meiner Betreuerin Petra Hofstedt, sowie den Mitarbeitern am Lehrstuhl für Programmiersprachen und Compilerbau an der BTU Cottbus bedanken. Ohne eure Unterstützung wäre diese Arbeit nicht zustande gekommen. Jeder Besuch bei euch in Cottbus hat mir Freude bereitet, sowohl die thematischen Diskussionen als auch die immer sehr menschliche Atmosphäre am Lehrstuhl.

Ein ganz herzlicher Dank auch an meine Schwester und Eltern sowie an meine Freunde. Anne, vielen Dank für die moralische Unterstützung beim Schreiben und die konstruktiven gemeinsamen Besuche in der SLUB. Chenzi für die regelmäßigen Mittagessenrunden und den Austausch über das Promotionsstudentenleben und Armin für die vielen unterhaltsamen Gespräche.

Weiterhin möchte ich all meinen Kollegen am Fraunhofer IIS/EAS danken. Hierbei insbesondere Thomas Klotz, Eva Fordran, Stephan Radke und Paul Ehrlich für die vielen Diskussionsrunden und die Ermutigungen weiterzumachen. Außerdem meinen, neuen, alten, Kollegen von COSEDA, welche mich gerade in der Endphase dieser Arbeit unterstützt haben.

Zusammenfassung

Vernetzte eingebettete Systeme bilden die Grundlage für das *Internet of Things* (IoT). Sie arbeiten häufig autark und in sicherheitskritischen Bereichen. Deshalb ist es wichtig sicherzustellen, dass die Systeme sich korrekt, also gemäß ihrer Spezifikation, verhalten. Zur Erstellung von Software für solche Systeme werden Betriebssysteme wie *Contiki* verwendet, welche die Programmierung von Anwendungen vereinfachen und die Portabilität zwischen verschiedenen Hardware-Plattformen ermöglichen.

In dieser Dissertation wird eine Methodik zur Verifikation von Software-Anwendungen, unter Berücksichtigung der Hardwareumgebung, für eingebettete Systeme anhand des Betriebssystems *Contiki* vorgestellt. Es wird die Technik des *Software Model Checking* [CGP00] – und dabei insbesondere *Bounded Model Checking* [BCC+03] - verwendet, welche es ermöglicht die Software eines eingebetteten Systems formal zu verifizieren.

Um die Verifikation durchzuführen, muss auch die Hardware des Systems modelliert werden. Die Herausforderung hierbei ist es, die Interaktionen der Software mit der Hardware hinreichend genau abzubilden und gleichzeitig den Aufwand für die Verifikation hinsichtlich des benötigten Rechenaufwands beherrschbar zu halten. In dieser Arbeit wird deshalb ein Ansatz verwendet, der die Treiber, die auf die Hardware zugreifen durch Softwaremodelle für die Verifikation ersetzt. Dies ermöglicht es Anwendungen für *Contiki* mit Hilfe einer abstrakten Verifikationsumgebung zu überprüfen.

Ein besonderer Aspekt bei eingebetteten Systemen ist die Verwendung von *Interrupts*, welche es ermöglichen Energie, einzusparen und auf externe Ereignisse zu reagieren. In bisherigen Ansätzen zur Verifikation werden Interrupts mit dem *Interleaving Modell* modelliert und das Verifikationsproblem mit Hilfe von *Partial Order Reduction* [CGP00, BK08] reduziert. Diese Arbeit zeigt, dass dieser Ansatz für die Verifikation von Anwendungen, welche periodische Interrupts verwenden, nicht ausreichend ist. Deshalb wird mit *Periodic Interrupt Modelling* ein neuer Ansatz zur Modellierung von Interrupts vorgestellt. Dieser Ansatz kann automatisiert angewendet werden und verringert die Anzahl von falschen Verifikationsergebnissen aufgrund ungenauer Modellierung. Zusätzlich ist es möglich Eigenschaften zu überprüfen, die von der Häufigkeit von Interrupt-Aufrufen abhängen.

Anhand des in der Arbeit entwickelten Verifikationsablaufs werden Beispielprogramme für *Contiki* untersucht und die Ansätze zur Interruptmodellierung verglichen.

Abstract

The main building blocks for the *internet of things* are connected embedded systems. Often these systems are also used in safety critical applications. Therefore, it is particularly important that these devices work according to their specification i.e. they behave as intended. Nowadays, even for simple devices embedded operating systems as *Contiki* are used to simplify application development and to increase portability between different hardware platforms.

The main objective of this thesis is to present a methodology for the verification of software applications written for the operation system *Contiki*, taking the system hardware into account. Therefore, *software model checking* [CGP00] and especially *bounded model checking* [BCC+03] is used as a technique, which allows to formally verify software for embedded systems.

For verifying the software against its specification, it is also necessary to build a model of the system hardware. Thereby, the difficulty is to create a model which is detailed enough to capture the hardware behavior so that the software performs correctly, while keeping the computation effort for the verification process manageable. In this work, the drivers which communicate with the hardware are therefore replaced with abstract models during the verification process. This enables the verification based on an abstract hardware platform independent of specific hardware.

A special role within embedded systems play *interrupts*. Interrupts are used to save power and can also be used to react on external events. Current methods for verification of interrupt driven software are based on the *interleaving model* and *partial order reduction* to reduce the size of the verification problem. This thesis argues that this method is not sufficient for software, whose behavior relies on periodically occurring interrupts. Therefore, in this thesis, a new approach called *periodic interrupt modeling* [CGP00, BK08] is introduced. This approach can be applied automatically and reduces the number of incorrect verification results due to inaccurate modeling. In addition, properties can be proven that depend on the number of occurring interrupts.

Using applications for the *Contiki* operating system, and based on a verification flow, the approaches toward interrupt modeling are compared.

Contents

1 Introduction and Scope **1**
1.1 Introduction 1
1.2 Summary of Contributions 3
1.3 Organization of the Thesis 4
1.4 Remarks on Notation, Commonly Used Definitions, and Terms 5

2 Related Work and Fundamentals **9**
2.1 Related Work 9
2.1.1 Simulation-Based Verification Techniques 9
2.1.2 Formal and Model Checking Based Verification Techniques . . . 13
2.1.3 Tools for Automatic Software Verification 16
2.2 Verification Using Model Checking 19
2.2.1 Model Checking Process 19
2.2.2 Model Checking Basics 20
2.2.3 Bounded Model Checking 23
2.2.4 Software Model Checking Using Bounded Model Checking and the CBMC Tool 27
2.2.5 Model Checking and Partial Order Reduction 32

3 The Operating System Contiki **39**
3.1 Introduction into *Contiki* 39
3.2 Example Application *LED Blink* 40
3.3 *Contiki* Kernel Scheduling Mechanism 42
3.3.1 Description of the Scheduling Algorithm 46
3.3.2 Scheduling of the *LED Blink* Example 48
3.4 Programming *Contiki* Applications 49
3.5 Hardware Access in *Contiki* 51

4 Modeling an Embedded System Running Contiki for Verification **55**
4.1 Overview of the Approach 55
4.2 Annotating Assertions for Verification 56
4.3 Modeling the System Environment Using Drivers 59

4.4 Interrupt Modeling 66
4.4.1 Existing Approaches 69
4.4.2 Applying Existing Approaches to the Verification of *Contiki* Applications 71
4.5 *Periodic Interrupt Modeling* - Taking System Runtime into Account . . 75

5 Model Checking and Verification Flow **79**
5.1 Bounded Model Checking and Setting Loop Bounds 79
5.1.1 Unbounded Loops in *Contiki* 80
5.1.2 Setting Bounds 83
5.1.3 Loop Unwinding and Interrupt Modeling 85
5.2 Verification Flow and Implemented Tools 86
5.2.1 Modeling and Specification Phase 86
5.2.2 Compilation and Interrupt Instrumentation 88
5.2.3 Verification Execution 89
5.3 Test Case Generation Using Bounded Model Checking 89

6 Verification of *Contiki* Applications **91**
6.1 Experimental Setup 92
6.2 *Hello World* Application 93
6.3 *LED Blink* Application 94
6.4 *LED Fader* Application 102
6.5 *Bubble sort* with LCD Application 104
6.6 *3-axis Acceleration Sensor with Rotation Detection* Application 108
6.7 Summary and Result Discussion 111

7 Conclusions **115**

A Appendix **119**
A.1 Example Applications Used for Evaluation 119
A.1.1 Transformed Source Code of Hello World Application for POR . 119
A.1.2 Transformed Source Code of Hello World Application for PIM . 119
A.1.3 Source Code of *LED Blink* Application Possibly Triggering an Event Queue Overflow 120
A.1.4 Source Code of LED Fader Application 121
A.1.5 Source Code of Bubble Sort with LCD Application 123
A.1.6 Source Code of 3-axis Acceleration Sensor with Rotation Detection Application 126
A.1.7 Source Code 3-axis Acceleration Sensor Declaration for *Contiki* 128
A.2 *Contiki* Kernel Implementation Details 131
A.2.1 *Contiki* Defined Kernel Events 131

A.2.2 *Contiki* Defined Process States 132
A.2.3 Process macros of *Contiki* 132
A.2.4 *Protothread* Macros of *Contiki* 133
A.2.5 *Contiki* Process API 133

Used Abbreviations **135**

List of Figures **137**

List of Tables **139**

List of Algorithms **141**

Bibliography **143**

1

Chapter 1

Introduction and Scope

In this chapter, an introduction to the topics of this thesis is given in Section 1.1 and the main contributions are summarized in Section 1.2. Afterwards, in Section 1.3, the further organization of this thesis is presented. Finally, in Section 1.4 general remarks about the used terms and notations are given.

1.1 Introduction

The use of connected *embedded systems* has increased rapidly in the last years, mainly due to the use of *Wireless Sensor Networks* (WSNs) [ASSC02] and the connection of more and more consumer goods in the *Internet of Things* (IoT) [AIM10].

Embedded systems are computing systems, which are designed to fulfill *specific* tasks *independently*, without direct interaction from a user. They are often embedded into other larger systems and can be applied in various application domains e.g. in medical devices, automotive systems or avionics. Due to their focus on a specific task, embedded systems are highly optimized and are characterized by a tight combination of hardware and software.

For programming of embedded systems often specialized operating systems like *Contiki* [Con17a, Wir14] are used. *Contiki* enables the development of low power applications that can be easily ported to different hardware platforms. In contrast to more well-known operating systems, including Linux or Windows, *Contiki* applications can run on microcontrollers that only consist of several kilobytes of RAM. Due to the requirements for low power consumption, the *Contiki* operating system is event-driven, i.e. events are used to trigger computations. These events can also be caused by *interrupts*, which allow it to save power and enable fast reaction times. For example, interrupts can wake up the system from low power processor modes or trigger special computa-

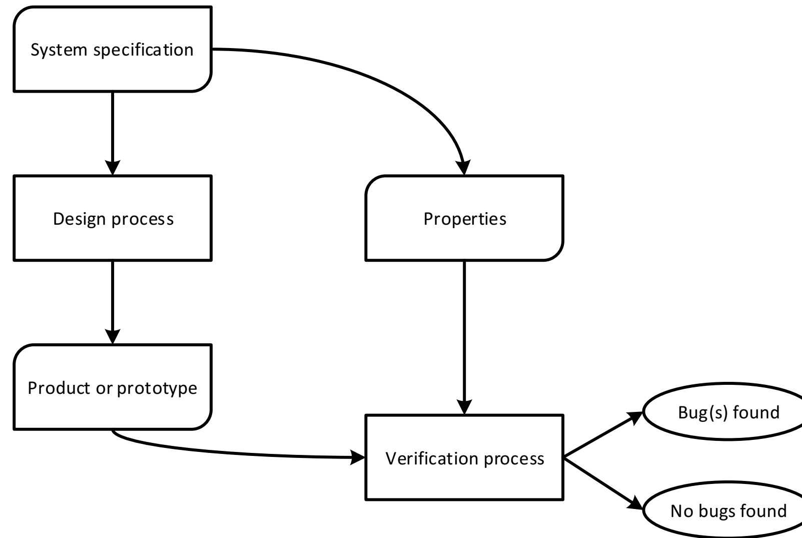


Figure 1.1: General system development and verification process applicable for embedded systems [BK08].

tions that are needed to react on external events, for instance, the arrival of new sensor values. Interrupt sources can also be hardware timers of the used microcontroller that enable periodic interrupts.

As embedded systems are also often used in safety critical environments the verification of the system behavior is very important. Verification in this context describes the process of checking that a system behaves as intended. In Figure 1.1 a general approach for designing and verifying systems is shown. Starting from a system specification, which describes the intended behavior of a to be developed system, a product is developed in a *design process*. On the other hand, in the *verification process*, it is made sure that the build system behaves as intended with regard to the original system specification. For verification two main concepts exist. Using *runtime* or *simulation-based* methods, the developed system (or a model of the system) is executed and observed to detect bugs or misbehavior. In contrast, *formal verification methods* rely on an analysis of the system structure and can be used to verify a system exhaustively, i.e. all possible system behaviors can be examined. One formal verification method is *model checking* that allows it to prove the correctness of a system automatically against a formal specification. Therefore, the system specification is translated into properties, which can be checked against a model of the system. A special kind of model checking is software model checking, which aims at the verification of software.

In this thesis, an approach to verify software applications for the operating system *Contiki* using software model checking is presented. The objective is the verification of *unmodified* software applications running on the system that are written in **C** and

are using the *protothread* programming model (part of the *Contiki* system). Problems which can be detected include common programming errors, e.g. arithmetic overflows, misuses of pointers, application-specific properties, and *Contiki* specific properties. Due to the tight interaction between hardware and software in embedded systems, also the hardware environment of the embedded system has to be considered and modeled to verify the software correctly. A correct modeling of the hardware is thereby essential and the main challenge. As noted by Baier and Katoen:

> "Any verification using model-based techniques is only as good as the model of the system [BK08]."

In contrast to other approaches, the application programming interfaces (APIs) of *Contiki* are used to model peripheral devices, e.g. sensors. By performing the verification at the level of the operating system it is possible to verify applications, which can be run on different hardware platforms. In addition to modeling the hardware, also interrupts have to be modeled as they influence software execution. The thesis shows, how different approaches for modeling interrupts affect the properties, which can be proven on a system. It is shown that current approaches for the modeling of interrupts are not sufficient to verify timing related properties for the operating system. Especially periodically occurring interrupts, which are typically caused by timers, have to be handled differently. Therefore, a modeling style called *periodic interrupt modeling* is introduced, to handle these kinds of interrupts.

To show the feasibility of the approach a verification platform for a *Contiki* based embedded system has been implemented, which allows it to compare the different modeling techniques. Several applications using this platform have been verified and results show the effect of different interrupt modeling techniques on verification time and the properties that can be proven. The results also demonstrate the limitations of the approaches, as well as of software model checking in general.

1.2 Summary of Contributions

The following list shows the main contributions of this thesis:

- A formalization of the event-driven *Contiki* kernel scheduling is given together with a description of the *protothreads* programming model.
- A modeling approach, which allows it to build an abstract verification platform for *Contiki* applications is introduced. This hardware modeling at the level of drivers allows it to verify *Contiki* applications independently of a specific hardware platform.
- For the modeling of interrupts existing techniques like *partial order reduction*

are introduced. It is discussed how well these are suited for the verification of *Contiki* applications especially for verifying timing related system behavior.

- A new technique for the verification of interrupts called *periodic interrupt modeling* is presented, which allows it to verify properties, that cannot be checked with existing modeling techniques due to over-abstractions of the actual system behavior.
- Based on the modeling techniques an automatic verification flow for *Contiki* applications has been implemented, based on the bounded model checking tool CBMC.
- Extensive verification results show the applicability of the modeling approach for the verification of *Contiki* example applications. Furthermore, the effect of the different modeling styles for interrupts regarding the overall verification effort is examined.

1.3 Organization of the Thesis

This thesis is structured as follows. In the following section of this chapter terms and notations used in this thesis are introduced. In Chapter 2 related work on the field of embedded system verification is presented. Therefore, simulation and formal-based verification approaches, as well as verification tools for software, are discussed. Afterwards, an introduction into model checking and the software model checking tool CBMC [CKL04, CBM17] is given.

Chapter 3 introduces the operating system *Contiki* using an example application. A formalization of the operating system kernel is given and the *protothread* programming model is introduced. Furthermore, it is shown how *Contiki* can be adapted to different hardware platforms.

The modeling approach used for verification is presented in Chapter 4. It is shown, which kind of properties shall be proven on the system and how they can be structured. Afterwards the modeling of the embedded system hardware is introduced. Thereby, a special focus is put on the modeling interrupts and the periodic interrupt modeling style is presented.

To perform verification using the CBMC tool, practical issues such as setting the number of loop unwindings have to be considered. Based on this an overall verification flow is developed, which is discussed in Chapter 5. In Chapter 6 verification results, obtained using the flow, for practical *Contiki* applications are shown.

Finally, in Chapter 7 conclusions are given.

1.4 Remarks on Notation, Commonly Used Definitions, and Terms

In this section, general notations and common definitions that will be used throughout this work are defined. The operator $\leftarrow$ is used to describe assignments to objects of any kind. For comparisons the operator $=$ is used. When not declared otherwise, the operators $\wedge$ (and), $\vee$ (or), $\implies$ (implication), $\oplus$ (exclusive or) and, $\neg$ (negation) are used as in Boolean algebra.

Definition 1.1. Let T be an arbitrary ***tuple*** $T = \langle t_1, t_2, ..., t_n\rangle$ with $n \in \mathbb{N}$ and $n > 0$ then $T.t_1$ denotes access to the first element of the tuple, $T.t_2$ to the second, $T.t_n$ to the n-th element.

Example 1.1. Let $P = \langle name, started\rangle$ be a tuple describing the state of a process, with $name$ being a string and $started \in \{true, false\}$. For a tuple $\langle calculate, false\rangle$ the tuple access operator can be used as follows: $\langle calculate, false\rangle.name = calculate$ and $\langle calculate, false\rangle.started = false$.

Definition 1.2. Given a $length \in \mathbb{N}$, $length > 0$, Q^{length} is a ***queue*** of arbitrary objects of the same type. $length$ defines the maximal number of objects that can be stored in the queue. A queue Q^{length} may contain n elements, written $Q^{length} = [q_0, ..., q_{n-1}]$, where $n \in \mathbb{N}$, $n \leq length$, i.e. they always reside in the first n positions of the queue. The following operations and notations are used when working with a queue:

- A queue is called *empty* when there are no elements in the queue.
- A queue is called *full* when there are *length* elements in the queue.
- The current number of elements in a queue can be accessed using $Q^{length}.size$[1].
- An element q can be added to the end of a queue using $Q^{length}.enqueue(q)$ when a queue is not full.
- An element can be removed from the beginning of a queue using $Q^{length}.dequeue$.
- The first element of a queue (q_0) can be accessed using $Q^{length}.first$ without removing it.
- The last element of a queue $((Q^{length}.size) - 1)$ can be accessed using $Q^{lengh}.last$ without removing it.

[1] The . operator is, in this case, a function application rather than a tuple access

Example 1.2. Let P be a tuple describing processes as described in Example 1.1, and let Q^4 be a queue of tuples P, which can store 4 elements at maximum. Let the queue contain two elements $Q^4 = [\langle calculate, false\rangle, \langle startup, true\rangle]$. The name of the first element can be accessed as follows $(Q^4.first).name = calculate$.

Another element can be added to the queue using the operation:

$$Q^4.enqueue(\langle shutdown, false\rangle)$$

The resulting queue is:

$$[\langle calculate, false\rangle, \langle startup, true\rangle, \langle shutdown, false\rangle]$$

Definition 1.3. Let S be al ***list*** of arbitrary objects of the same type $S = [s_0, s_1, ..., s_n]$ with $n \geq 0$. The following operations and notations are used when working with a list:

- A list is called *empty* when there are no elements in the list.
- An element in the list can be accessed by its position in the list (*index*) and angle brackets, e.g. $S\langle 0\rangle$ returns s_0.
- An element s can be added to the beginning of the list by $S.prepend(s)$ incrementing the index of all the elements already in the list *when it is not already in the list*.
- An element s can be removed from the list by $S.remove(s)$ decrementing the index of the subsequent elements in the list.

Example 1.3. Let P be a tuple describing processes as described in Example 1.1, and Let S be an list of tuples P. The list contains the elements $S = [\langle calculate, false\rangle, \langle startup, true\rangle]$. Another element can be added to the list using the operation $S.prepend(\langle shutdown, false\rangle)$. The resulting list is:

$$[\langle shutdown, false\rangle, \langle calculate, false\rangle, \langle startup, true\rangle]$$

An element can be removed from the list using the operation $S.remove(\langle calculate, false\rangle)$. The resulting list is:

$$[\langle shutdown, false\rangle, \langle startup, true\rangle]$$

Definition 1.4. Let M be a set of objects in memory. The address p of an object

$m \in M$ can be stored in a ***pointer***. The following operations and notations are used when working with pointers:

- The operation $*p$ returns the object $m \in M$ to which p points to (dereference operator).
- The operation $p \mapsto b$ accesses b as part of the object $m \in M$ to which p points to. It is a short form of $(*p).b$.
- In case a pointer p does not point to any object it has the value *null*.

Example 1.4. Let p be an pointer to the queue of elements described in Example 1.2. The elements can then be accesses as $((*p).first).name = calculate$ or $(p \mapsto first).name = calculate$.

An ***interrupt*** is used to modify the control flow of a program running on a processor of a system. Interrupts are caused by certain events in the system. Whereby, an interrupt which is caused by the software running on a processor is called ***software interrupt*** and an interrupt caused by the hardware is called ***hardware interrupt***. When an interrupt occurs, the current state of the system execution is stored and a designated function, which is associated with the type of the interrupt, is executed. Such a function is called ***interrupt handler*** or ***interrupt service routine*** (ISR) When this function returns, execution of the code at the stored location resumes.

A ***software interrupt*** allows it to react to certain conditions that occur in a program by calling a special instruction, which invokes the ISR. The advantage of using a software interrupt is that its faster than a normal function call, as it is supported by the processor. As software interrupts are deterministic and caused by software, no special modeling for this kind of interrupts is needed. Furthermore, *Contiki* doesn't use this kind of interrupts. Therefore, software interrupts are not further examined in this thesis.

A ***hardware interrupt*** is caused by an asynchronous event and pauses the execution of arbitrary code running on a processor when a defined hardware event occurs, e.g. the overflow of a timer register or the change of a designated pin.

Depending on its type, interrupts can be enabled and disabled by software running on the system (interrupt masking). Furthermore, interrupts can have priorities, where interrupts with higher priority can pause the execution of interrupts with lower priority. Further details on classes of interrupts and their use in microcontrollers can be found in [Bäh10].

Definition 1.5. A ***process*** or ***thread*** is an entity of code belonging to an application, which is to be executed sequentially on an operating system.

Depending on the kind of computing system, threads can be run in parallel or pseudo parallel. An operating system ***kernel*** has the purpose to start and select processes.

In this thesis, the terms ***property*** and ***assertion*** are used to denote a formalized part of a system specification using temporal and Boolean algebra. Properties can be classified into different types, whereas with ***assertions*** such safety properties (see Section 2.2.2) are denoted, which are supported by the CBMC tool and can be written using a **C** `assert` statement.

Chapter 2

2 Related Work and Fundamentals

This chapter summarizes related work and presents fundamentals that are used in this thesis. Different modeling and verification techniques can be applied to low power systems targeted in this work and especially to the software running on them. Therefore, related work on simulation-based (Section 2.1.1) and model checking based formal verification techniques (Section 2.1.2) is discussed. Furthermore, in Section 2.1.3 other formal verification techniques and tools are introduced.

In this thesis, model checking is applied for verification. Therefore, the general model checking process (Section 2.2.1) and basic terms (Section 2.2.2) are introduced. A special model checking technique, which can also be applied to software, is bounded model checking. A summary is given in Section 2.2.3 and the tool CBMC for the verification of **C** programs using this technique, is presented in Section 2.2.4. To reduce the size of the model checking problem when targeting systems with parallel executions, partial order reduction can be used. An introduction to this technique is given in Section 2.2.5.

2.1 Related Work

2.1.1 Simulation-Based Verification Techniques

Simulation-based approaches for embedded system software verification are based on executing the actual software. This can be either done on the real hardware of the embedded system, or on a virtual simulation platform of the hardware. During simulation, the software is executed for pre-defined system inputs (a test case), and the

system behavior is monitored to check that the software behaves as expected. The biggest advantage of a simulation is that very large systems can be handled. The general limitation of all simulation based approaches is that only a certain execution trace of the software is examined. The correctness of the software execution is often determined by evaluating the results, or by comparing them with expected values derived from the system specification. The validity of the simulation based approach is largely dependent on:

- the quality of the test cases executed,
- the abstraction level of the underlying hardware model, when running on a simulation platform,
- the ability to detect and debug the software on real hardware.

The straightforward approach for testing embedded system software is running it on the target system. This approach has the advantage that no additional bugs are introduced due to the inaccurate modeling of the system hardware. However, this approach is also problematic, as occurring bugs can often not be reproduced and debugging can become cumbersome as not all internal states of the system (especially external hardware such as sensors) can be accessed.

As an alternative, when developing software for an operating system for embedded systems like *Contiki* or TinyOS [LMP+05], dedicated simulators can be used for the operating system. The COOJA simulator [ODE+06] was originally developed for the simulation of *Contiki* based sensor networks, whereas TOSSIM [LLWC03] can be used for TinyOS-based systems. These simulators are discrete event simulators, which focus on the simulation of several sensor nodes. They allow it to optimize communication and network algorithms for e.g. low power consumption or cases when sensor nodes or network connections fail. However, the actual hardware platform of a sensor node is often not considered. The TOSSIM simulator, for example, doesn't capture a specific CPU instruction set architecture. The COOJA simulator, however, can be coupled with the instruction set simulator MSPSim [EDF+07], which supports the TI MSP430 class of processors. MSPSim is written in Java and allows it to describe peripheral devices of a sensor node, such as sensors LEDs or radio communication. Using MSPSim and COOJA together a complete WSN can be simulated at the level of the instruction set of an MSP430 processor. By using this approach it is also possible to simulate sensor nodes running different operating systems together, such as *Contiki* and TinyOS [EÖF+09].

In [JZD09] a survey comparing different network simulators for complete WSNs can be found. The strong point of the described simulators for WSNs is the simulation of the interplay of different sensor nodes and of their network behavior. However, for performance reasons the behavior of the sensor node hardware is abstracted.

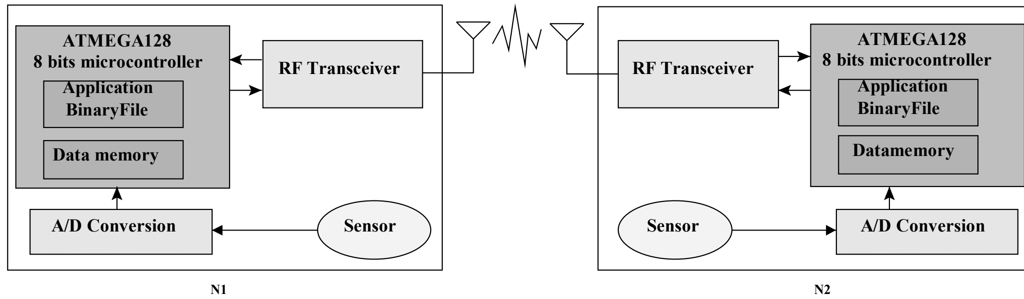


Figure 2.1: Wireless sensor network modeled using SystemC AMS [VPB+08].

The hardware of embedded systems can be modeled more accurately using system description languages like SystemC [IEE12, MRR03], which provide different abstraction levels and modeling styles. As SystemC is an extension library for C++, software can be easily integrated into the system. Especially by using *transaction level modeling* fast virtual prototypes can be built, which allow to run the software of the system during simulation of the hardware. Using the SystemC AMS extensions [IEE16], it is also possible to model analog mixed-signal (AMS) behavior, e.g. needed for the accurate description of sensor behavior. In [VPB+08] a SystemC AMS model of a WSN containing two nodes is presented. Each node contains an RF-Transceiver, an ATMEGA 128 Bit microcontroller and a sensor including analog/digital conversion (shown in Figure 2.1). For modeling the microcontroller a cycle-accurate instruction set simulator is used, allowing it run arbitrary applications. However, as the focus of the work is modeling of the RF-transmission, only a simple application is considered, especially no operating system is considered.

Another example of the use of SystemC for WSNs is shown in [HDG+09]. The objective of this work is to model the overall power consumption of a sensor network. In this work also an instruction set simulator and state machines are used to model the behavior of the hardware of a single sensor node. To model the communication between nodes a TLM based communication scheme is used, inspired by the network simulator OMNeT++ [Var01].

The so far described approaches mainly focus on the simulation of embedded systems, but not on methodical verification, i.e. making sure that the system behaves as intended with regards to the specification or for the detection of bugs. When simulating a design often certain scenarios are tested and repeated to make sure that the system doesn't contain any faults. The correct behavior can be specified by the addition of run-time assertions to the code. In [NWE+07] a validation tool for TinyOS is described, which automates the adding of assertions. This approach is also called *runtime validation*, as errors are detected when running the system to prevent catastrophic behavior. Using a special compiler the system software is annotated with checks for e.g.

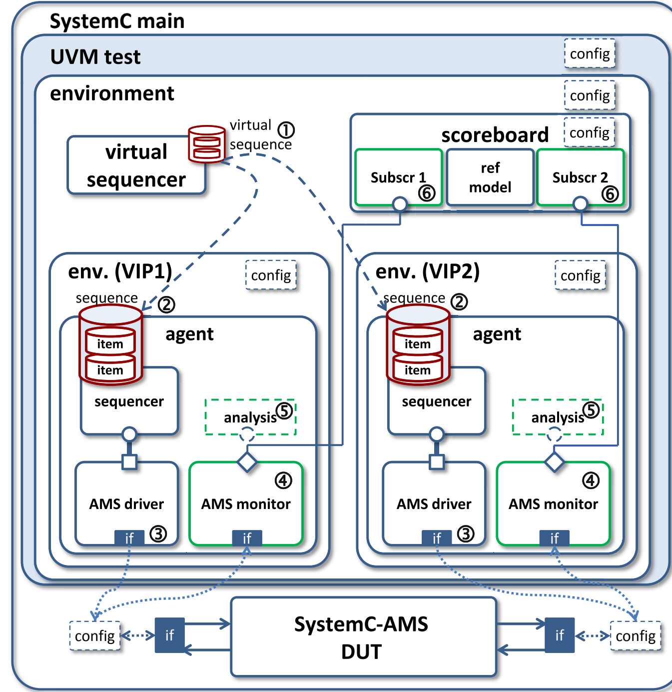


Figure 2.2: A UVM testbench for the structured verification of a SystemC design [VKE+14].

arithmetic overflows during software execution. If such a situation occurs, an error handler is called. As this approach modifies the original code, a runtime overhead exists. A similar approach for Contiki is shown in [PK09]. In these two works the approach is considered to be used on real hardware deployed in the field. The main advantage is that errors can be detected, which are introduced by failing hardware. However, it can also be useful for running software using a hardware simulation model to detect programming errors (e.g. by assuming a wrong range of sensor values).

In the field of hardware verification (mostly for digital circuits) so-called *verification methodologies* have been established, due to the required higher quality standards with regards to bugs, as the hardware cannot be modified after production. Furthermore, due to the high verification effort, there is a necessity to re-use code of already existing verification environments (also called *test benches*). The in the industry applied *universal verification methodology* (UVM) [Acc14] is such a verification methodology. UVM is based on the coverage-driven verification principle introduced in the *e* verification language [IEE11] and has been originally developed to verify digital circuits.

In the UVM methodology, the creation of test benches is standardized. Figure 2.2 shows the structure of a complete UVM test environment. The main principle of UVM is to apply randomly generated stimulus as input to the *design under test* (DUT),

which is the system that is verified. Using so-called *functional coverage* points it is checked, how many parts of the original specification are covered by the test scenarios, i.e. it is checked whether all parts of the system are triggered by the test scenario. UVM is based on similar abstractions principles as used for the modeling of complex systems using transactions to abstract specific timing behavior. The overall test scenario is described by *sequences*. Sequences are composed of *sequence items*[2], which contain data that is sent to the DUT. A sequence item can be e.g. the values received from an external sensor. The contents of sequences and sequence items are generated randomly and shaped using constraints to valid values within the specification. A *driver* translates the sequence items into the protocol that the DUT supports and sends them to the DUT.

A similar principle is used to extract data from the DUT. A *monitor* component examines signals of the DUT and extracts data that is then stored in a sequence item. The overall check, whether a test scenario was executed successfully is done in a so-called *scoreboard*. Therefore, the data that has been sent to the DUT via drivers and the data received from monitors is sent to the scoreboard and there compared using a reference model.

To structure the verification environment, reusable *verification components* (also called verification intellectual property (VIP)) are introduced that encapsulate behavior, which can be reused. A verification component constains a driver, monitor, and sequences, which are protocol specific. To control several VIPs *virtual sequences* are used that start other sequence. By layering sequences, the abstraction level of the test case description can be raised.

In recent years UVM has been made also available in SystemC to support system-level verification of combined hardware-software systems [BPV14]. Additional effort has been made to support also constraints and coverage in SystemC [HLGD12, VKE+14]. Application examples for automotive embedded systems using UVM in SystemC can be found in [BDE+15]. UVM-SystemC is currently an active research topic. However, due to its roots in digital hardware verification, further effort is still needed to fully support the verification of software.

[2] A UVM sequence item is similar to a SystemC transaction. A transaction describes data content that is transmitted between system components, without specifying details of the transmission protocol.

2.1.2 Formal and Model Checking Based Verification Techniques

Formal verification techniques are in general not based on the execution of the actual system. They are based on analyzing the structure of the system checking the validity of the specification. Especially, model checking (see Section 2.2) has seen a widespread industrial use for the verification of synchronous digital circuits, due to its fully automatic nature and high-quality standards required for those circuits. Therefore, many commercial tools are available, which allow it to verify implementation level (register-transfer-level RTL) code, written in languages such as VHDL [IEE09] or SystemVerilog [IEE13]. The formalization of the system specification is mainly done using domain specific property specification languages such as *Property Specification Language* (PSL) [IEE10] and *SystemVerilog Assertions* (SVA) [IEE13]. These specification languages simplify the writing of complex properties by introducing additional constructs not available in LTL or CTL (see Section 2.2), e.g. referring to the stability of a signal. However, due to model checking inherent limitations regarding the maximum state space of the examined system, it is often restricted to the verification of critical submodules of an overall design.

Formal verification for SystemC designs, which can be used to model embedded systems has been an ongoing research topic since the first releases of SystemC. An overview of the challenges that arise when verifying SystemC compared to traditional formal hardware verification approaches is given in [Var07]. In general, approaches can be separated into more HW specific approaches (SystemC is used more like a digital hardware description language) and more towards higher systems abstraction levels, including software. For example in [DG05] a first approach is described to transform a subset of SystemC into a finite state machine representation, on which bounded model checking is performed. However, due to the transformation process it is restricted to a subset of C++ and not suited for the verification of software. For example, pointers are not supported in the transformation process.

Approaches to verify SystemC by transforming it into other representations are numerous and can be found in e.g.:

- In [Her10] the transformation of SystemC designs into Timed Automata is presented, which can be checked using the UPPAAL [LPY97] model checker.
- Another representation to capture the behavior of SystemC is to use *Abstract State Machines* as shown in [OHT04]. Using these Abstract State Machines a PCI bus is modeled, whereas properties are formulated using PSL. The state machines are then verified using the standard SMV model checker [SMV17]. Afterwards, the state machines are translated into SystemC. The backward trans-

lation of SystemC into abstract state machines is shown in [HT05].

- In [KS05] an approach is described, where it is tried to automatically detect which parts of the SystemC description relate to hardware and which parts describe software. Based on these parts different abstraction and optimization techniques are applied to verify the system. The model checking tools SATABS [CKSY05] together with SMV [SMV17] are used to verify the system. In contrast to the other approaches, a larger subset of the **C++** language is supported by the approach.
- The verification of high-level untimed SystemC transaction-level models based on the model checking tool CBMC is presented in [GLD10]. Property specification is done using a version of PSL, which is extended with language constructs needed for high-level SystemC verification. To tackle the problem of bounded model checking being not complete when dealing with unrestricted loops (see Section 2.2.4), an induction based approach is introduced, which allows to prove completeness for safety properties.
- A comprehensive analysis of several transformations approaches for SystemC, which allow it to compare different model checking tools is described in [CNR13]. Three general approaches are presented. Firstly, the transformation into a finite-state-machine representation, which is checked using the *SPIN* model checker [Hol04]. Secondly, the transformation into a sequential program. Thirdly, the transformation into a multi-threaded program.

  Especially it is shown how the non-deterministic SystemC scheduler can be mapped, so that also possible interleavings of the thread execution of SystemC threads can be checked. Using their approach ESST (explicit-scheduler/symbolic threads) significant improvements over a sequential mapping of threads is shown. This approach is implemented in the KRATOS model checker.

The general challenge when verifying practical SystemC designs is that SystemC is based on C++ and imposes no restrictions on the use of external libraries. Therefore, often the SystemC design needs to be transformed to work together with a specific model checking tool. Furthermore, all parts of the system have to be available in a format, which can be analyzed by the model checker (e.g. as source code).

Another approach to formal software verification for embedded systems is to verify the source code of the program without using an explicit hardware model. Therefore, the hardware part of the system is described in an abstract way so that the software can still be verified. When verifying the software, the exact timing of external hardware such as sensors is, for example, neglected.

In [LFCJ09] a verification approach is presented that combines formal and non-formal

verification techniques. As starting point, test cases are written to check whether the overall software works as intended. For formal verification, the software is split up into hardware-dependent and hardware-independent parts. This is necessary as no embedded operating system is considered, which provides an abstraction layer to the hardware. Abstract models of the system hardware are considered to verify also timing behavior. For this purpose, the SMV language is used. To perform the verification several model checking tools (CBMC used in this work, SATABS, NuSMV) are compared. As an example, a medical embedded system based on an Atmel AT89S8252 (microcontroller with an 8051-like architecture) system is used, which contains typical components such as sensors, serial interfaces, and timers. In their work, however, no operating systems are considered.

In [MVÖ+10] the model checking tool Anquiro is presented. Anquiro supports the formal verification of Contiki applications at different abstraction levels starting from a hardware specific model of a sensor node to a system-wide model, which includes a network of sensor nodes. The **C** code is translated into an FSM representation, which is handled by the *Bogor* model checker, properties are specified using LTL. Anquiro's main focus is the verification of network applications running on several nodes and includes constructs for network communication. No details are given how more hardware related properties can be checked. The authors suggest to couple their approach with a hardware model, but no verification results are given.

An approach to the verification of Linux device drivers together with hardware models is given in [HTV+13]. Their approach is based on a CBMC extension which supports multiple threads. The goal is to find bugs in device drivers by checking them against models of hardware devices. These models are written in **C** and are manually extracted from the QEMU hardware emulator, which provides a library of models. These drivers are annotated with properties, which the check for correct usage of the hardware by the driver. To model the hardware and driver software, different threads are used, which can preempt each other, allowing the hardware model to cause interrupts in the driver. The work describes in detail how a real timer clock, temperature sensor using the I^2C protocol, and an Ethernet MAC are modeled and gives corresponding results.

2.1.3 Tools for Automatic Software Verification

For the verification of software, many tools exist, especially in academia. In [DKW08] a survey of different software verification techniques is presented. In Table 2.1 a list of formal verification tools is presented, based on the techniques model checking, abstract interpretation, and symbolic execution. Most tools for software verification, especially for embedded systems work on the **C** language, however, the supported language constructs differ, especially with regard to the support of pointers and bit-level

Tool	Developer	Technique	Input language	Property Specification
CBMC	CMU/Oxford University	Bounded model checking	**C** [1]	`assert` statements
SATABS	Oxford University	Abstraction based model checking	**C** [1]	`assert` statements
CPA Checker	Sosy-Lab/University Passau	Abstraction based model checking / Configurable solvers	**C** [2]	`assert` statements
SLAM/Static Driver Verifier	Microsoft	Abstraction based model checking	**C**	Predefined rules
Kratos	Fondazione Bruno Kessler	Abstraction based model checking	SystemC/**C** [3]	`assert` statements
Bogor	Kansas State University	Model checking / Configurable state space search	Bogor modeling language	`assert` statements
Java Path Finder	NASA	Explicit state model checker	Java Byte Code	`assert` statements + configurable rules
Astrée	LIENS/CNRS/INRIA	Abstract interpretation	**C**	`assert` statements
Klee	Stanford University	Symbolic Execution	C++, LLVM bitcode	`assert` statements

[1] Complete ANSI **C** standard, C++ subset supported, floating point support
[2] Subset of GNU-C, no details specified [CPA17]
[3] No support for pointers, pointer arithmetic, dynamic object creation, enumerations, bit precise operations [CAA+11]

Table 2.1: Overview on verification tools for software verification.

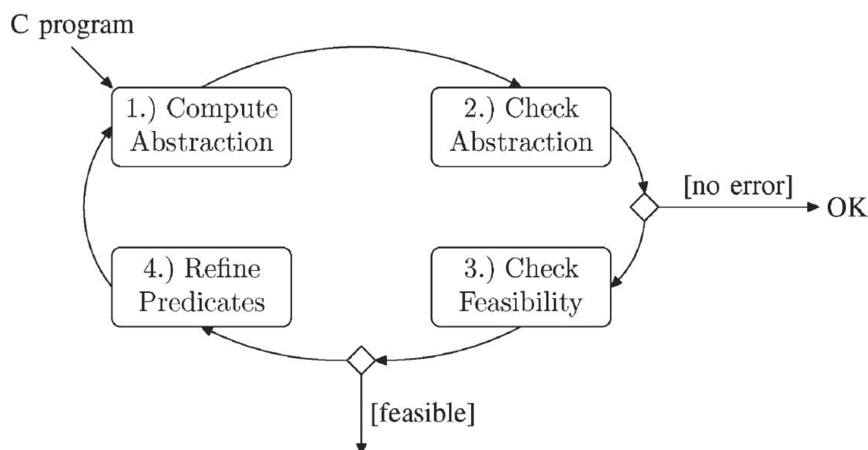


Figure 2.3: Principle approach for counterexample guided abstraction refinement [DKW08].

arithmetic. The properties that shall be verified are mostly expressed as safety properties using `assert` statements (cf. Section 2.2.4). The following paragraph summarizes the techniques.

Verification based on model checking Model Checking and bounded model checking is presented in more detail in the following Section 2.2. A short summary of the model checking based tools follows: Satabs [CKSY05], CPAChecker [BK11a], SLAM [BLR11], and KRATOS [CAA+11] are all based on the CEGAR (counterexample guided abstraction refinement) principle, where an abstraction of the actual program is calculated. The basic principle is shown in Figure 2.3. As the state space of a program is often too large to be searched, it is tried to prove properties on an abstraction of the program. In a first step, this abstraction is calculated. In a second step, it is tried to prove the property on the abstracted description. The abstraction is chosen to be sound, that means when the property is valid on the abstracted description, it is valid also on the original system. Therefore, when a property is valid it holds also for the original program and verification is successful. When the verification fails (step three) it is checked, whether the generated counterexample also is valid on the original program (it is checked whether the counterexample is feasible cf. Figure 2.3). If this is the case, a valid counterexample was found on the program. However, when the counterexample does not lead to an error, the abstraction has to be refined (step four), as it leads to a counterexample not existing on the real system. This refinement step takes the failed property into account to calculate a new abstraction. This abstraction loop is implemented automatically. However, finding a suitable abstraction for a program is not a trivial task. Therefore, the mentioned tools implement different abstraction strategies, according to the problem. Further model

checking tools for software, which do not use **C** as input language are e.g. the Bogor model checker [RDH03], which uses an own modeling language (BIR), allowing it to model at a suitable abstraction level e.g. by defining custom data types. Bogor is designed as a framework, allowing it to implement different model checking algorithms. Another model checking tool is the Java Path Finder [VHB+03] tool, developed at NASA, which can be used to verify Java programs. It checks Java bytecode using its own Java virtual machine. Originally based on the Spin model checker [Hol04], it has been steadily developed into a separate tool. In contrast to most other software verification tools, it is an explicit state space model checking tool, similar to the original Spin model checker.

Verification based on abstract interpretation Compared to model checking, *abstract interpretation* [CC77] trades in accuracy i.e. false alarms can occur, where no actual bugs exist within the program to reach higher verification performance. For *abstract interpretation* an abstract program semantic is used to check whether a program can reach a forbidden state.

A tool which implements *abstract interpretation* for **C** programs is Astrée [CCF+09]. It was developed to verify software for Airbus. The tool is sound with respect to the defined properties, meaning when it finds no bugs, the software can be considered bug free. However, *false positives* can occur. A *false positive* is a problem or bug found by a verification tool, which is caused by modeling inaccuracies and does not exist on the real system.

Verification based on symbolic execution Another technique that has been successfully applied to verify programs is *symbolic execution* [Kin76]. In contrast to model checking and abstract interpretation, the goal is not to verify a program completely but to rather automatically generate test cases, which cover a large part of the system and led to a high code coverage. To speed up performance, not all possible program executions are considered, which can lead to incomplete results, i.e. bugs are not found. A tool which implements symbolic execution for **C** programs is Klee [CDE08], which has been able to find deep bugs in the *GNU COREUTILS*. Klee supports the Low Level Virtual Machine (LLVM) [LA04] assembly language and therefore supports the complete C++ language standard using the `clang` compiler, part of the LLVM infrastructure. Bounded Model Checking tools like CBMC can also be used to automatically generate test cases as shown in [VK12].

2.2 Verification Using Model Checking

2.2.1 Model Checking Process

Model checking is a variant of formal verification, which allows it to verify systems automatically and completely, with respect to the given specification of the system, using a so-called *model checking tool*. To apply model checking [BK08] separates the model checking process into three phases:

- In the *modeling phase*, the system under consideration is modeled using the input language of the used model checking tool. This step can be done automatically, by the translating description of the system e.g. from a hardware description language as VHDL or a programming language such as **C**. However, manual interaction is still in many cases necessary. When systems are too large to be handled by the model checking tool often manual abstractions are necessary to reduce the system size. Furthermore, the system description is often not complete with respect to the environment needed for correct operation, making refinements necessary to avoid *false positives*.

  The second part of the modeling phase consists of formalizing the natural language specification of the expected system behavior into a specification language, supported by the model checking tool. Thereby, the specification is translated into several properties (also called assertions), which shall be verified on the system.

- In the *running phase*, the model checking tool is executed and the actual model checking algorithm is performed. The model checking algorithm will return, whether the formal specification is valid (holds) on the system or a counterexample exists. Furthermore, it can happen that running the model checking does not terminate, as the state space of the system, which is examined, is too large to be handled and the tool runs out of memory.

- In the *analysis phase*, the results returned from the model checker are analyzed. When the checked properties pass as valid, it has to be analyzed whether further properties exist that need to be proven. When a property fails, the returned counterexample has to be analyzed to identify the cause of the failing property, which can either be a failure in the system model or the checked property.

  In case the system is too large to be handled, the size of the state space has to be reduced by e.g. abstracting or removing parts of the system not relevant for proving a property or using compositional verification techniques.

2.2.2 Model Checking Basics

In general model checking has been defined on so-called Kripke structures as described in [CGP00]:

Definition 2.1. Let AP be a set of atomic propositions. A ***Kripke structure*** M over AP is a four-tuple $M = \langle S, I, T, L \rangle$ where

- S is a finite set of states,
- $I \subseteq S$ is the set of initial states,
- $T \subseteq S \times S$ is a transition relation that must be left-total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $T(s, s')$,
- $L : S \to 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

Definition 2.2. A ***path*** in the structure M from a state s is an infinite sequence of states $\pi = s_0 s_1 s_2 ...$ such that $s_0 = s$ and $T(s_i, s_{i+1})$ for all $i \geq 0$. Furthermore, we denote $\pi(i) = s_i$ and with $\pi^i = (s_i, s_{i+1}, ...)$ the suffix of π beginning in s_i.

To formulate properties (specification of the system), which shall be proven on a Kripke structure temporal logics like LTL (linear temporal logics), CTL (computation tree logic) or CTL* can be used. These logics differ in their expressiveness, whereas CTL* is the most expressive, but the most difficult to use and harder to implement. LTL and CTL are the subsets of CTL* which are commonly used. Which logic is the best fit for a problem lead to great debates in the model checking community and is greatly dependent on the problem [Hol04]. For the upcoming explanations, LTL is used.

The syntax of LTL is defined on Boolean variables and uses the standard operators of Boolean algebra ($\wedge$, $\vee$, $\neg$) . In addition, temporal operators have been added such as **X** (”next time”), **F** (”eventually”), **G** (”globally”), **U** (”until”) and **R** (”release”). The semantics of LTL are defined over paths of a Kripke structure. In Figure 2.4 simple examples for the use of temporal operators of LTL are given.

The formal definition of the LTL semantics is as follows [BCC+03].

Definition 2.3. Let M be a Kripke structure, π a path in M and f be an LTL formula. Then $\pi \models f$ (f is valid along π) is defined as follows.

- $\pi \models p$ iff $p \in L(\pi(0))$
- $\pi \models \neg f$ iff $\pi \not\models f$
- $\pi \models p \wedge f$ iff $\pi \models p$ and $\pi \models f$

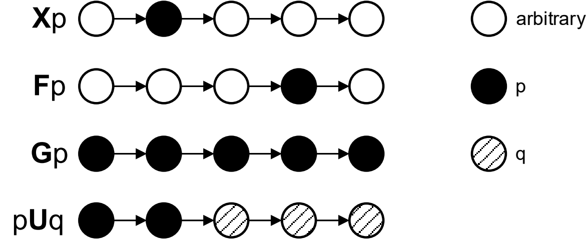


Figure 2.4: Visualization of the LTL operators and their semantics.

- $\pi \vDash p \vee f$ iff $\pi \vDash p$ or $\pi \vDash f$
- $\pi \vDash \mathbf{X}f$ iff $\pi^1 \vDash f$
- $\pi \vDash \mathbf{G}f$ iff $\pi^i \vDash f$ for all $i \geq 0$
- $\pi \vDash \mathbf{F}f$ iff $\pi^i \vDash f$ for some $i \geq 0$
- $\pi \vDash f\mathbf{U}g$ iff $\pi^i \vDash g$ for some $i \geq 0$ and $\pi^j \vDash f$ for all $0 \leq j < i$
- $\pi \vDash f\mathbf{R}g$ iff $\pi^i \vDash g$ if for all $j < i, \pi^j \nvDash f$

The properties which are used for system specification can be differentiated by the kind of specification that they convey. They are often differentiated into:

- *reachability properties* - a certain state of the system can be reached,
- *safety properties* - something bad never occurs,
- *liveness properties* - something good will occur,
- *fairness properties* - something good will occur infinitely often.

In this work, the software model checking tool CBMC is used, which supports the verification of safety properties of the kind $\mathbf{G}f$, where f does not contain further temporal operators. Further details on LTL and examples for the different types of properties can be found in [BMB+01].

The Model Checking problem for LTL formulas can then be defined as:

Definition 2.4. A temporal formula f holds on a Kripke structure M, $M \vDash f$, iff for all paths π starting in I of M holds $\pi \vDash f$.

Example 2.1. As an example for the basic formalization used in model checking, Figure 2.5 shows the Kripke structure for a modulo 4 counter. Starting from an initial value of 0, the system counts repeatedly from 0 to 3. The Boolean variables x and y represent the binary value of the counter in the system. The Kripke structure $M_{Mod4Counter}$ using the atomic propositions $AP = \{x, y\}$ corresponding to this system

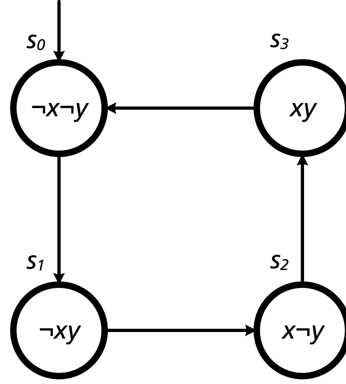


Figure 2.5: Kripke structure for a modulo 4 counter.

can be formulated as:

$$
\begin{aligned}
&M_{Mod4Counter} = \langle S, I, T, L\rangle \text{ with:}\\
&S = \{s_0, s_1, s_2, s_3\}\\
&I = \{s_0\}\\
&T = \{(s_0, s_1), (s_1, s_2), (s_2, s_3), (s_3, s_0)\}\\
&L(s_0) = \{\emptyset\}, L(s_1) = \{y\}, L(s_2) = \{x\}, L(s_3) = \{xy\}
\end{aligned}
$$

A safety property for this system would be $\mathbf{G}(\neg x \vee \neg y)$, which checks that either x or y are always false. A model checking algorithm tries to find, in an efficient manner, whether a path exists for which the property is violated. In this case, it is checked, whether a path through the system exists where x and y can become true.

The classical approach to solving the LTL model checking problem [VW86], is based on the observation that both a Kripke structure M and formula f can be represented by a corresponding *Büchi automaton* A (denoted as A_M and A_f). The set of infinite words accepted by a Büchi automaton is denoted as $\mathcal{L}(A)$. Model checking then corresponds to computing that $\mathcal{L}(A_M) \bigcap \overline{\mathcal{L}(A_f)}$ is empty i.e. that all words accepted by A_M are also accepted by A_f .

One of the biggest problems when applying model checking in practice is the so-called *state space explosion* problem, as in model checking the state space of a system is examined. In Example 2.1 the number of states was 4, however the state space of a 32-bit variable is 2^{32} states, which need to be checked.

Therefore, different algorithms have been implemented for tackling the state explosion problem. One of the most important achievements was the introduction of *symbolic model checking* [BCM+90] implemented in the *Symbolic Model Verifier (SMV)* [SMV17] tool. Symbolic model checking does not store the explicit state graph. Rather

binary decision diagrams (BDDs) are used as underlying data structure allowing significantly larger systems to be checked.

A further improvement based on *symbolic model checking* is *bounded model checking* (BMC), which represents states symbolically but does not use BDDs. The model checker CBMC [CKL04, CBM17], which is used in this thesis, is based on the BMC principle.

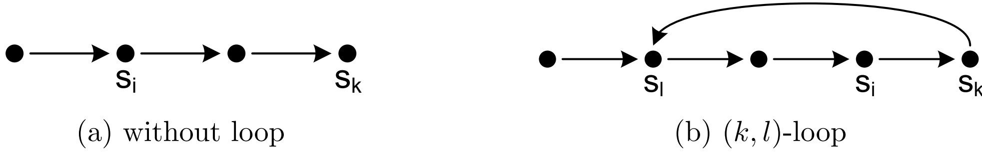
2.2.3 Bounded Model Checking

BMC [BCCZ99, BCC+03] tackles the state space explosion problem by transforming the model checking problem into a *Boolean satisfiability* (SAT) problem. SAT solvers [BHvM09] are used in different domains and although being an NP-complete problem, very large instances can be practically solved. A further change is that BMC only searches the state space up to a certain bound k from the initial states of a system. When applying BMC this bound k is increased either till

- the property is violated (leading to a shortest path violating the property),
- the problem has become so large that it cannot be handled,
- a *completeness threshold* has been reached, meaning that the whole state space of the system has been searched.

The calculation of a completeness threshold is a hard problem and often practically not feasible. Further details can be found in [BCC+03]. As a completeness threshold can often not be reached, BMC is mostly used for bug hunting i.e. the absence of errors for a certain bound can only be shown. However, as the bound k is increased step by step the size of the verification problem is also increased gradually, and as a consequence generated counterexample traces are minimal. Furthermore, the bound k gives a good intuition how deep the system is searched. In digital circuits, the bound can be seen as the number of clock cycles that a system runs from a reset state. When verifying software it can be seen as the number of program statements the program runs or how many loops are executed (see Section 2.2.4).

In the following, a sketch of the BMC algorithm and its encoding into a SAT problem is shown. An LTL formula is said to be valid, if it holds on all paths of a Kripke structure. Therefore, for showing that a formula is not valid, it needs to be proven that there exists at least one path from an initial state which violates it. Although LTL formulas are defined over infinite paths, a bounded path can be used to represent infinite behavior, when a back loop exists. Therefore the notion of a so-called (k, l)-loop is introduced, which is also shown in Figure 2.13. Figure 2.6(a) shows a path without a loop, whereas in Figure 2.6(b) a path is shown that contains such a loop.



(a) without loop (b) (k, l)-loop

Figure 2.6: Bounded paths without and with loop [BCC+03].

Definition 2.5. For $l < k$ we call a path π a (k, l)-loop if $T(\pi(k), \pi(l))$ and $\pi = u \cdot v^{\omega}$ (v^{w} denotes an infinite repetition of v) with $u = (\pi(0), ..., \pi(l-1))$ and $v = (\pi(l), ..., \pi(k))$. We call π a k-loop if there exists $k \geq l \geq 0$ for which π is a (k, l)-loop.

Whether an LTL formula holds on a bounded path depends whether this path π contains a k-loop. When π is a k-loop ($k \geq 0$) an LTL formula f is valid along a bounded path iff $\pi \vDash f$, i.e. the semantics corresponds to those defined in Definition 2.5. However, when the path does not contain a k-loop a special bounded semantics is needed. In this semantics for a bounded path without a loop, a formula $\mathbf{G}f$ is always false, as it is not possible to make a prediction for the infinite behavior. The detailed bounded semantics for paths with a loop and without a loop can be found in [BCCZ99, BCC+03].

Based on the bounded semantics of LTL and the unwinding of a Kripke Structure, BMC can be formulated as a SAT problem by the following equations. $[M]_k$ is used to unwind the Kripke structure to a certain depth k:

$$[M]_k = s_0 \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \text{ with } s_0 \in I \tag{2.1}$$

To translate an LTL formula it has to be checked whether a back loop exists on a path, therefore, L_k is used;

$$L_k = \vee_{l=0}^{k} {}_lL_k \text{ with } {}_lL_k = T(s_k, s_l) \tag{2.2}$$

Using this loop condition an LTL formula f can be mapped into a propositional formula. The translation $[f]_k^i$ thereby represents the translation of f for paths without a loop and ${}_l[f]_k^i$ for those with a loop. The detailed recursive definition is given in [BCCZ99, BCC+03]. Using these translations the overall BMC SAT problem is defined as follows:

$$SAT?\,[M, f]_k = [M]_k \wedge \left(\left(\neg L_k \wedge [f]_k^0\right) \vee \bigvee_{l=0}^{k} \left({}_lL_k \wedge {}_l[f]_k^0\right) \right) \tag{2.3}$$

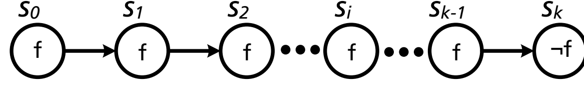


Figure 2.7: BMC principle for formulas of safety properties of the kind $\mathbf{G}f$.

When a satisfying assignment for this formula is found, this assignment represents a witness for a path and is, therefore, a counterexample.

When verifying *safety properties* of the kind $\mathbf{G}f$ as described in Section 2.2.2, BMC can be seen a searching a path to a state, which does not satisfy f (a path $\mathbf{F}\neg f$ is searched), as shown in Figure 2.7. In this case Formula 2.3 can be simplified to:

$$\text{SAT? } s_0 \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg f(s_i) \text{ with } s_0 \in I \tag{2.4}$$

As just the existence of a bad state reachable from the initial state is searched, no special condition is needed to check for the existence of loops within the system.

Example 2.2. Based on the modulo 4 counter introduced in Example 2.1 it can now be demonstrated, how a SAT formula is built. The property $\mathbf{G}\neg(x \wedge y)$ shall be checked on the modulo counter (meaning that counter can never reach the value 3, binary "11"). The following equations describe the initial state and the transition relation symbolically:

$$s_0 \in I : (\neg x_0 \wedge \neg y_0)$$
$$T(s_i, s_{i+1}) : ((x_{i+1} = (x_i \oplus y_i)) \wedge (y_{i+1} = \neg y_i))$$

As a safety property is checked, it needs to be checked that a state can be reached that violates the property:

$$f : \neg(x \wedge y)$$

and $\neg f$ encoded as:

$$\neg f(s_i) : (x_i \wedge y_i)$$

Using Formula 2.4 the SAT problem can be built. The bound k is incremented till

a counterexample is found or the search depth is assumed big enough. For $k = 3$ a satisfying assignment can be found for this example, which correspond to the following SAT formula:

$$
\begin{aligned}
&s_0 \in I : (\neg x_0 \wedge \neg y_0) \wedge \\
&T(s_0, s_1) : ((x_1 = (x_0 \oplus y_0)) \wedge (y_1 = \neg y_0)) \wedge \\
&T(s_1, s_2) : ((x_2 = (x_1 \oplus y_1)) \wedge (y_2 = \neg y_1)) \wedge \\
&T(s_2, s_3) : ((x_3 = (x_2 \oplus y_2)) \wedge (y_3 = \neg y_2)) \wedge \\
&\neg f(s_0) : (x_0 \wedge y_0) \vee \\
&\neg f(s_1) : (x_1 \wedge y_1) \vee \\
&\neg f(s_2) : (x_2 \wedge y_2) \vee \\
&\neg f(s_3) : (x_3 \wedge y_3)
\end{aligned}
$$

When passing this formula to a SAT solver, the following satisfying assignment is calculated for the variables x_i, y_i:

$$
\begin{aligned}
x_0 &= \text{False}, y_0 = \text{False} \\
x_1 &= \text{False}, y_1 = \text{True} \\
x_2 &= \text{True}, y_2 = \text{False} \\
x_3 &= \text{True}, y_3 = \text{True}
\end{aligned}
$$

This assignment corresponds to a path from the initial state s_0 via s_1 and s_2 to the property violating state s_3.

2.2.4 Software Model Checking Using Bounded Model Checking and the CBMC Tool

Bounded Model Checking is most successfully applied for model checking for digital synchronous circuits, however, it has also shown to be useful for the verification of software. One of the first applications was to check the equivalence of hardware descriptions in the hardware description language Verilog compared with a software specification written in **C** [CK03]. Out of this approach for translating **C** programs into a SAT formula, the model checker CBMC [CKL04, CBM17] was developed, comparing Verilog designs with **C** implementations is still possible using the extension HW-CBMC.

Modeling and Specification Using CBMC

The CBMC model checker has since its first release been steadily been developed. The tool is available as open source and available for Linux, Windows, MacOS platforms It can be used as a command line replacement for typical compilers such as GCC [GCC17]. CBMC supports thereby the complete ISO/IEC 9899:1999 **C** language standard, including bit-vector arithmetic and support for floating point computations.

As CBMC is a software model checking tool, the input to the tool is the source code of an application, written in **C**. The application that shall be checked has to be completely available as source code, with no pre-compiled libraries being allowed. However, CBMC supports parts of the *C standard library*, for which an implementation is provided. An important part of software model checking is to model the environment of the application such as user inputs or feedback from the hardware the application is running on. Therefore, CBMC supports the construct of non-deterministic variables:

```
int x = nondet_int();
```

The variable `x` is assigned an arbitrary value from the range of integer variables. To restrict the input to a subset of values an assume statement can be used:

```
__CPROVER_assume(x>=0 && x<=10);
```

In this case, the variable `x` is restricted to the interval 0..10.

To describe the specification of a program `assert` statements are used, which are a kind of safety properties, stating that the described Boolean condition must hold for all possible executions of the program. For example

```
assert(x != 0);
```

checks that `x` never takes the value 0, at the point the `assert` statement is executed. In the following, the safety properties supported by CBMC shall be called *assertions*.

In addition to user-defined assertions, CBMC is able to add assertions automatically by static analysis of the code. These assertions include checks for common programming errors in **C** such as violation of array bounds, dereferencing of invalid pointers, division by zero, and checks for arithmetic overflows. These automatically generated assertions can be activated as an option for verification when running CBMC.

Translation into a SAT Formula

This section sketches how the BMC principle is applied for software, based on the steps performed by the model checker CBMC for **C** programs. An overview about

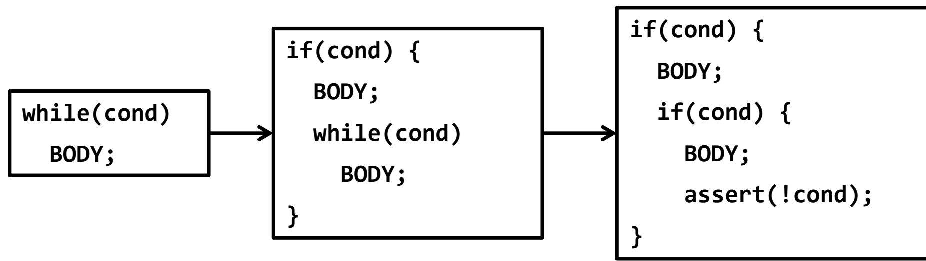


Figure 2.8: Loop unwinding as done for BMC and adding of unwinding assertions.

the application of BMC and SAT for software verification can be found in [BHvM09]. The translation and encodings performed by CBMC are described in more detail in [KCY03]. CBMC applies the following steps to convert a **C** program into a BMC problem:

- The control flow of the program is simplified and side effects are removed.
- Loops within the program are unwound to a certain depth similar to BMC.
- Conversion of the resulting program into a SAT formula.
- SAT solving and converting the solution into a counterexample.

Control flow simplification Before the program is encoded into a sat formula, some simplifications on the control flow are performed. First, the program is brought into a standard format, whereby side effects are removed by semantically equivalent constructs. For example `k=j++;` is transformed into `k=j; j=j+1;`. Control flow statements such as `continue`, `break`, `return`, as well as `switch` statements, are transformed into a form that only uses `if` and `goto` statements. Loop statements such as `for` and `do while` are replaced by `while` loops.

Loop unwinding To perform BMC, loops inside a program are unwound to a certain depth. The principle applied, is shown in Figure 2.8. The `while` loop is replaced by copying the body of the loop and guarding it with an `if` statement. This process is performed a user definable number of times (called *unwinding depth*) for each loop in the program. This depth is similar the setting of the bound k used in BMC. In general, it is not possible to automatically set an unwinding depth and therefore has to be done by the user. However, by adding so-called *unwinding assertions* (`assert` statement shown on the right-hand side of Figure 2.8), it is possible to check whether enough unwinding of the program has been done. These assertions are automatically added by CBMC for each unwound loop in the program. When a program passes verification and no unwinding assertion is violated it can be assumed fully verified. Otherwise, the unwinding depth for the violating assertion is incremented till it passes

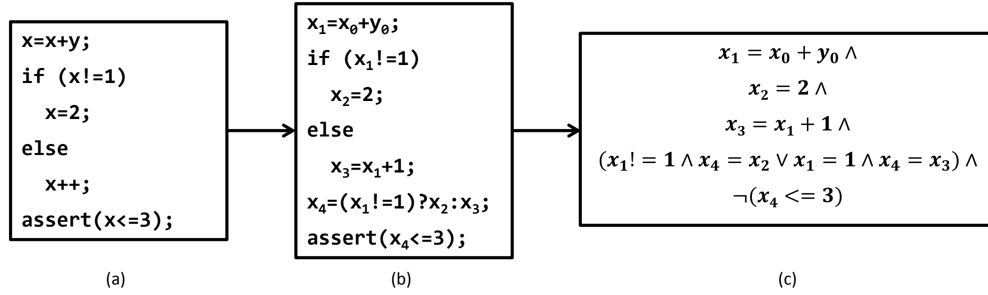


Figure 2.9: Example of translating a simplified program into a SAT formula [CKL04].

verification. The so determined unwinding depth can also be seen as possible worst case execution time of the program to be run.

The described approach, however, is only possible for applications, that shall terminate. Embedded system software is usually written in such a way that it shall not terminate i.e. it contains a `while (1)` loop as part of the program. In this case, BMC can be used for bug hunting only.

Conversion to a SAT formula The simplified and loop free program code can then be encoded straight forward into a SAT formula. This process is shown exemplary in Figure 2.9 The original program code (a) is first transformed into a *static single assignment* form (b). In this form for each assignment towards a variable a new variable of the same type is created (denoted by an index for each assignment), which is from now on used for reading. Special care has to be taken when the code branches at an `if` statement. A case select has to be added which chooses the valid variables depending on the condition of the original statement (x_4 in the example). The static single assignment form is transformed by conjugating each statement ((c) in Figure 2.9), whereby `assert` statements are negated as in BMC. The encodings used for other statements, especially pointers, arrays, memory allocation and jumps can be found in the literature.

Before the resulting formula can be passed to the SAT solver all mathematical operators part of the formula have to be resolved. This is done by so-called *bit-blasting*. Therefore operators are replaced by Boolean bit-level arithmetic as used in digital circuits and synthesis tools, where the operations are mapped to basic gates [Jor04]. These mappings depend on the hardware architecture used, especially regarding the width of variables, floating point operations, and pointers. These widths can be adjusted in the CBMC tool and tuned so that they represent the target hardware the application runs on. In addition to SAT solvers, satisfiability modulo theory (SMT) solvers can be used, which allow direct reasoning over bit-vector arithmetics [BHvM09].

SAT solving and counterexample generation The resulting formula is passed to a SAT engine, whereby the default SAT solver of CBMC is the MiniSAT solver [Nik17], which is also used for this work. Other solvers, (e.g. SMT solvers) are also supported by CBMC, a comparison of these solvers is however out of scope for this work.

The SAT solver generates a satisfying assignment when a counterexample exists, which assigns a value for each variable of the program in static single assignment form. This assignment is converted into a counterexample trace (see Example 2.3). For each statement in the program, it is shown, which value gets assigned to each variable during each statement in the program.

Example 2.3. This example, shows how the modulo 4 counterexample (Example 2.2) can be modeled using CBMC. Figure 2.10(a) shows the corresponding source code. The counter is realized using the integer variable `count`, which is incremented in a `while` loop. The property to be checked in the example is shown in line 5 and is similar to the property used in Example 2.2. It is checked that the counter variable never takes the value 3 (binary value ”11”). When verifying this property the unwind bound parameter for CBMC has to be set to at least the value 3 for the `while` loop, as otherwise the automatically generated unwinding assertions are triggered. For this unwinding parameter a counterexample exists, which is shown in Figure 2.10(b). The counterexample starts in the initial state[3] of the program (line 3, `counter = 0`) and shows all the assignments in the program till the property is violated (line 15). Using the counterexample it is possible to see all states the program runs through, leading to the violation of the property.

2.2.5 Model Checking and Partial Order Reduction

The biggest problem when verifying systems using model checking is the size of the state space. Especially, the size grows when examining parallel programs which share variables for communication. One technique to reduce the state space in such systems is the *partial order reduction* (POR), also known as model checking using representatives. This section gives a short introduction to the POR principles following the description as given by Clarke et al. in [CGP00], as well as by Baier and Katoen in [BK08].

> “A common model for representing concurrent software is the *interleaving model*, in which all of the events in a single execution are arranged in a linear order called an interleaving sequence. Concurrently executed events

[3] Some internal CBMC states, initializing the program thread, have been omitted from the trace.

```
1 int main(void) {
2   unsigned int count=0;
3   while (1) {
4     count = (count+1)%4;
5     assert(count != 3);
6   }
7 }
```

(a) **C** model `count.c` of the counter and specification

```
1  State 15 file count.c line 2 function main thread 0
2  ----------------------------------------------------
3    count=0u (00000000000000000000000000000000)
4
5  State 17 file count.c line 4 function main thread 0
6  ----------------------------------------------------
7    count=1u (00000000000000000000000000000001)
8
9  State 20 file count.c line 4 function main thread 0
10 ----------------------------------------------------
11   count=2u (00000000000000000000000000000010)
12
13 State 23 file count.c line 4 function main thread 0
14 ----------------------------------------------------
15   count=3u (00000000000000000000000000000011)
16
17 Violated property:
18   file count.c line 5 function main
19   assertion
20   count != (unsigned int)3
21
22 VERIFICATION FAILED
```

(b) Excerpt from counterexample trace

Figure 2.10: Modulo 4 counterexample modeled for CBMC.

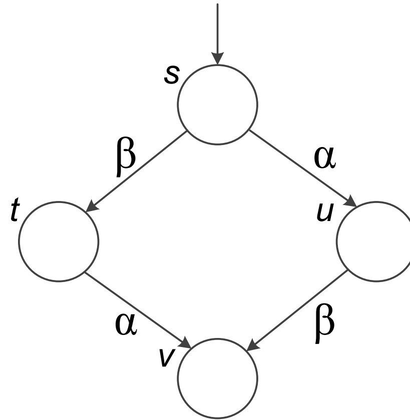


Figure 2.11: Interleaving diamond for α and β [BK08, CGP00].

> appear arbitrarily ordered with respect to one another, (...) all possible interleavings of such events are normally considered. This can result in an extremely large state space." [CGP00]

The basic idea of POR can be thus be given as:

> "When a specification cannot distinguish between two interleaving sequences that differ only by the order in which concurrently executed events are taken, it is sufficient to analyze only one of them." [CGP00]

A general example of the idea of reducing interleavings is given in [BK08] in the following way:

Example 2.4. Let $P1$ and $P2$ be programs that are executed in parallel. $P1$ contains the assignment $x \leftarrow x + 1$ and $P2$ contains the assignment $y \leftarrow y - 3$, whereby x and y are local variables of the corresponding programs. Furthermore α and β denote the assignment in $P1$ and $P2$. The reachable states of the system constructed using the interleaving model are shown in Figure 2.11. It can be seen that state v can be reached independently of the order in which α and β are executed. Thus, when only state v is relevant to a property for model checking, only one ordering must be considered.

A more formal description of the general concepts of POR follows, based on the definitions in [CGP00].

Definition 2.6. A ***state transition system*** is a quadruple $\langle S, T_s, I, L\rangle$ where the set of states S, the set of initial states I, and the labeling function L are as defined as for a Kripke structure (see Definition 2.1) and T_s is a set of transitions such that for each $\alpha \in T_s, \alpha \subseteq S \times S$.

A Kripke structure $M = \langle S, I, T, L\rangle$ may be obtained by defining T so that $T(s, s')$ holds when there exists a transition $\alpha \in T_s$ such that $\alpha(s, s')$.

Definition 2.7. For a transition $\alpha \in T_s$ we say that α is ***enabled*** in a state s if there is a state s' such that $\alpha(s, s')$ holds. The set of transitions s is $enabled(s)$.

In the example in Figure 2.11 $enabled(s) = \{\alpha, \beta\}$. Furthermore, only deterministic transitions will be considered. A transition is *deterministic*, when for a state s there exists at most one state s' such that $\alpha(s, s')$.

The goal of POR is now to calculate a set $ample(s) \subseteq enabled(s)$, which is used for model checking instead of $enabled(s)$, while preserving the correctness of the model checking algorithm. Therefore, the following goals are formulated both in [CGP00] and [BK08] to calculate $ample(s)$:

- The set $ample(s)$ must have sufficient behavior so that model checking is still correct, with respect to the checked property.
- The set $ample(s)$ should be considerably smaller than $enabled(s)$.
- The overhead for calculating $ample(s)$ should be small so that a calculation makes sense, compared to checking $enabled(s)$.

In other words, the goal of POR is to construct a reduced Kripke structure for interleaved parallel executions, which fulfills the same properties as the original one.

Before it will be described how transitions for a set *ample(s)* can be selected, some further definitions are given. An *independence* relation for transtitions can be defined as:

Definition 2.8. An ***independence*** relation $I_n \subseteq T_s \times T_s$ is a symmetric, antireflexive relation, satisfying the following two conditions for each state $s \in S$ and for each $(\alpha, \beta) \in I_n$

- *Enabledness*: If $\alpha, \beta \in enabled(s)$ then $\alpha \in enabled(\beta(s))$.
- *Commutativity*: $\alpha, \beta \in enabled(s)$ then $\alpha(\beta(s)) = \beta(\alpha(s))$.

Definition 2.9. The ***dependency*** relation D is the complement of I_n: $D = (T_s \times T_s) \setminus I_n$.

Based on this definition it can be seen that the execution of the transitions in Figure 2.11 is independent.

Within Kripke structures each state is labeled with a set of atomic propositions AP (see Definition 2.1).

Definition 2.10. A transition is ***invisible*** when its execution from any state does not change the value of the propositional variables in AP. A transition is *visible* when it is not invisible.

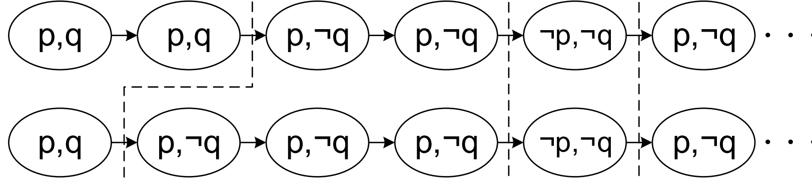


Figure 2.12: Two stuttering equivalent paths [CGP00].

Related to the visibility of transitions is the concept of *stuttering equivalent* paths.

Definition 2.11. Two ***paths*** σ and ρ are ***stuttering equivalent***, denoted by $\sigma \sim_{st} \rho$, when each path can be partitioned into blocks such that states in the k-th block of σ are labeled the same way as in the k-th block ρ.

An example of this is shown in the Figure 2.12. The two paths are stuttering equivalent, as the paths can be partitioned into blocks (although of different length), so that the order of the labels of the states in the blocks is the same. It can be easily seen that next operator **X** of the temporal logic LTL (see Section 2.2.2) must be removed from specifications so that two stuttering equivalent paths cannot be distinguished anymore. A property of this kind is called *invariant under stuttering*. Because of this, a subset of LTL is defined in the following way:

Definition 2.12. The subset of the logic LTL without the next operator **X** is denoted by $\mathbf{LTL}_{\neg\mathbf{X}}$.

It has to be noted, that the safety properties mostly used in software model checking (and as supported by the CBMC tool) are of the kind $\mathbf{G}f$, i.e. they are in $\mathbf{LTL}_{\neg\mathbf{X}}$ and thus invariant under stuttering.

Definition 2.13. Two ***structures*** M and M' are ***stuttering equivalent*** if and only if:

- M and M' have the same set of initial states.
- For each path σ of M that starts from an initial state s of M there exists a path σ' of M' from the same initial state s so that $\sigma \sim_{st} \sigma'$.
- For each path σ' of M' that starts from an initial state s of M' there exists a path σ of M from the same initial state s so that $\sigma' \sim_{st} \sigma$.

The following sentence expresses that an $\mathbf{LTL}_{\neg\mathbf{X}}$ property cannot distinguish two stuttering equivalent structures: Let M and M' be two stuttering equivalent structures. Then for every $\mathbf{LTL}_{\neg\mathbf{X}}$ property f and every initial state $s \in I, M, s \models f$ if and only if $M', s \models f$.

As an example for stuttering equivalence when in Figure 2.11 transition α is invisible, then $L(s) = L(u)$ and $L(t) = L(v)$ and thus $stv \sim_{st} suv$.

Under the assumption of a specification that is expressed as an $\mathbf{LTL_{\neg X}}$ property, a set *ample*(s) can now be constructed, which preserves the correctness of the formula and is then used for model checking. For constructing a set as noted in [BK08] there exist *static* and *dynamic* approaches. The *static* approach is based on calculating *ample*(s) statically based on the original structure and reducing it (This approach is applied in this thesis as described in Section 4.4). In contrast, the *dynamic* approach reduces the transitions on the fly during the execution of a model checking algorithm. Therefore, instead of giving a concrete algorithm tailored to the specific model checking approach for selecting $amples(s) \subseteq enabled(s)$ when checking $\mathbf{LTL_{\neg X}}$ properties, the following four conditions have to be fulfilled. Thereby, these conditions use the above given definitions for dependence and visibility.

- **C0**: $ample(s) = \emptyset$ iff $enabled(s) = \emptyset$ (nonemptiness condition)
- **C1**: Along every path of the full-state graph starting in s, a transition dependent on a transition α from *ample*(s) must be preceded by α (dependency condition).
- **C2**: If $ample(s) \neq enabled(s)$ then every $\alpha \in ample(s)$ is invisible (invisibility condition).
- **C3**: A cycle is not allowed if it contains a state in which some transition α is enabled, but is never included in *ample*(s) for any state s on the cycle (cycle condition).

Condition **C0** states that a state which has a successor in the original structure must also have a successor in the reduced structure. The most important condition is **C1**, which makes sure that all dependent transitions remain in *ample*(s) and are executed in the correct order. The conditions **C2** and **C3** make sure that the generated structures are stuttering equivalent. Especially **C2** states that *ample*(s) only contains transitions, which are invisible and therefore only lead to states with the same labeling.

To illustrate the conditions [CGP00] presents several examples. For **C1** and **C2** the following examples are given:

Example 2.5. Consider again the interleaving diamond in Figure 2.11 and selecting α as *ample*(s) and therefore removing state t from the system. Because of condition **C2**, α must be invisible and therefore the path $stv \sim_{st} suv$.

Example 2.6. Consider again the interleaving diamond in Figure 2.11 and assume there is a transition γ enabled from t as shown in Figure 2.13(a). To select again α as *ample*(s) the following has to hold so that it is possible to remove the state t to construct the reduced structure. First, γ must be independent of α otherwise **C1**

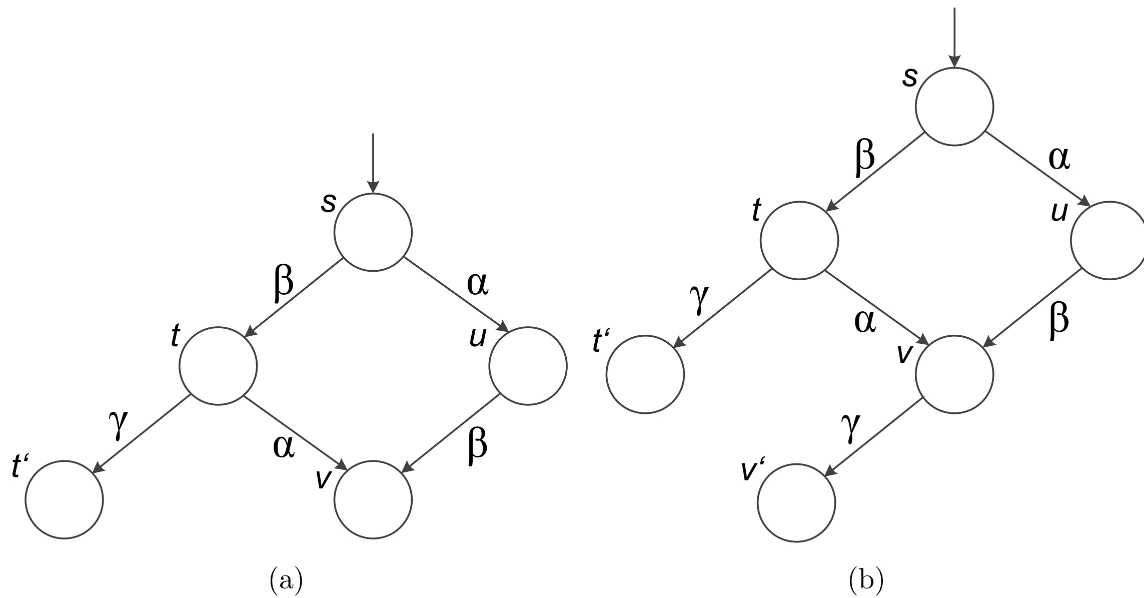


Figure 2.13: Extended interleaving diamond with additional transition, illustrating Example 2.6.

would be violated, which states that a transition dependent on a must be executed before α. Second, because γ is independent of α, it must be enabled in v as shown in Figure 2.13(b) and therefore leading to a state v'. Third, due to condition **C2**, α must be invisible and therefore $stt' \sim_{st} suvv'$. Therefore, an $\mathbf{LTL}_{\neg\mathbf{X}}$ property cannot distinguish the two state sequences.

Further explanations and justifications for the conditions and the consequences on model checking performance can be both found in [CGP00] and [BK08].

Chapter 3
3 The Operating System Contiki

The goal of this thesis is to verify applications written for the embedded operating system *Contiki*. Therefore, in this chapter, the *Contiki* operating system for embedded systems is introduced. A general introduction into Contiki is given in Section 3.1. Afterwards, Section 3.2 shows a typical example of a *Contiki* application, and Section 3.3 describes the *Contiki* operating system kernel semantics. Furthermore, *Contiki* introduces its own programming model for applications, which is described in Section 3.4. An important aspect of *Contiki* is its portability to different hardware platforms. The access to hardware is summarized in Section 3.5.

3.1 Introduction into *Contiki*

Contiki [Con17a] is an open source operating system for embedded systems, e.g., nodes of wireless sensor networks. *Contiki* makes it possible to connect very low power embedded devices as used for *Internet of Things* (IoT) applications.

The development of the operating system started in 2002 under the lead of Adam Dunkels at the Swedish Institute of Computer Science (SICS) with the goal of developing an operating system for low power embedded devices. Since then it has been steadily developed and has evolved. It was originally based on uIP, a full TCP/IP stack, which can run on 8-bit microcontrollers and was also created by Dunkels. In contrast to more widely known operating systems like Linux or Windows, it can run on devices with only a few kilobytes of RAM. Due to this low power consumption, *Contiki* was also ported to devices such as the Apple IIe and Commodore 64 [Wir14].

To reach the goal of low power consumption and a low memory footprint, *Contiki*

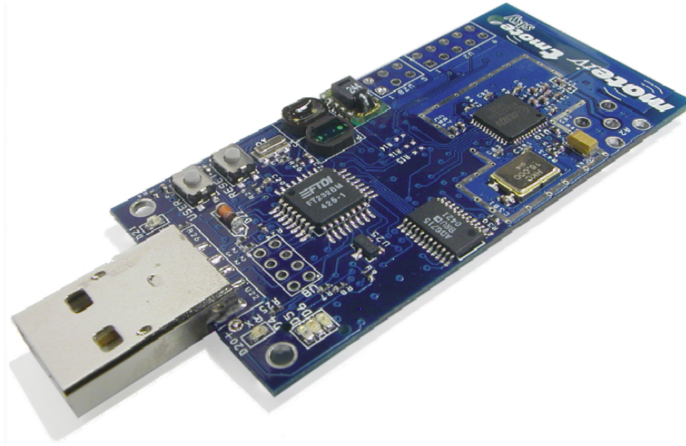


Figure 3.1: *Tmote sky* wireless sensor node as used for *Contiki* [TMo07].

has an event-driven kernel. Furthermore, *Contiki* has the goal to be easily portable to different hardware platforms by introducing APIs for standard embedded system devices. As *Contiki* is written completely in **C** it can be easily compiled for different processor platforms. To ease the development of applications the *protothread* programming model (see Section 3.2 and 3.4) was introduced in 2005, which allows it to easily write portable applications consisting of several threads.

Hardware platforms for *Contiki* based embedded systems consist of microcontrollers such as the TI MSP430 [MSP17] or Atmel AVR [AVR17] (further supported processors are listed at [Con17b]) and a number of application-specific sensors, which are connected using protocols such as SPI [SPI16] or I^2C [Sem14]. Furthermore, *Contiki* supports standard sensor nodes such as the *tmote sky* platform depicted in Figure 3.1.

3.2 Example Application *LED Blink*

Contiki applications are written in **C** using the *protothread* programming model that consists of a set of **C** macros that allow the programming of *Contiki* applications in a thread (or process) like fashion, making it easier to write event-driven programs. A detailed description of the underlying scheduling mechanisms and of the programming model is given in Section 3.3 and Section 3.4.

An example of an application written for the *Contiki* operating system is shown in Figure 3.2. After a certain period of time, this application turns on and off LEDs connected to the embedded system. The shown application is part of the official *Contiki* release as an example for different hardware platforms that support LEDs

```
1  PROCESS(blink_process, "Blink");
2  AUTOSTART_PROCESSES(&blink_process);
3  PROCESS_THREAD(blink_process, ev, data)
4  {
5    PROCESS_EXITHANDLER(goto exit;)
6    PROCESS_BEGIN();
7    while(1) {
8      static struct etimer et;
9      etimer_set(&et, CLOCK_SECOND);
10     PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
11     leds_on(LEDS_ALL);
12     etimer_set(&et, CLOCK_SECOND);
13     PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
14     leds_off(LEDS_ALL);
15   }
16   exit:
17     leds_off(LEDS_ALL);
18     PROCESS_END();
19 }
```

Figure 3.2: Contiki example *LED Blink* application using *protothreads*.

[Con17a]. It demonstrates how applications are programmed, the event timer system of *Contiki* works, and how access to peripheral devices of the system platform is possible via *Contiki* provided APIs.

A detailed description of the example follows[4]. In line 1 a name and a string (used when printing information) are assigned to the application process (`blink_process` and `"Blink"`). In line 2 the application is registered to be started automatically at system startup of *Contiki* using the macro `AUTOSTART_PROCESSES`. The actual definition of the thread containing the code of the application starts in line 3. The parameters are the name of the process the thread belongs to (`blink_process`), as well as the event (`ev`) and data item used (`data`) when a thread is invoked by the *Contiki* kernel using an event (see Section 3.3).

As part of the process in line 5, an exit handler is defined, which is called when the process is invoked using the `PROCESS_EVENT_EXIT` event, signaling that the process should exit. This call could be initiated by the *Contiki* kernel or another application. In this case, the program jumps to line 16 and turns off the LEDs of the system when the *LED Blink* application ends.

One of the features of *protothreads* is that they allow suspending and resuming of processes at the location where they have given up control to the *Contiki* kernel. The macros `PROCESS_BEGIN` and `PROCESS_END` (lines 6 and 18) are used to show such areas.

[4] **C** macros are written in uppercase letters.

Code outside of these areas is executed every time a process is invoked, regardless where it was suspended. This is also the reason why the exit handler is placed outside of the macros defining beginning and end of a process.

The turning on and turning off of the LEDs is done in an unbounded while loop defined in lines 7 to 15. Inside the loop in line 8, an event timer `et` is defined. It is defined as static, as Contiki doesn't save the stack during suspending a process. Therefore all variables which shall be used after resuming a process must be declared using the **C** `static` keyword.

After each second the LED shall be turned on and off. Therefore, the event timer `et` is registered with the event timer system of *Contiki* (line 9). The macro `CLOCK_SECOND` is platform specific and tells the event timer system when a second has passed, e.g. in ticks of a hardware timer register. The macro `PROCESS_WAIT_EVENT_UNTIL` (line 10) is used to suspend the process until a condition has been fulfilled. During that time other processes in the system can run. When the application process gets invoked by an event it is checked whether the condition, that the event time has expired - meaning that one second has passed - has been fulfilled. In line 11 the LED gets turned on, using the *Contiki* provided API `leds_on` and the macro `LEDS_ALL`, which has to be implemented for each hardware platform. This process is repeated in lines 12-14 to turn the LEDs off.

Although the application is running in an unbounded loop, it is not blocking the system, as it always gives control to the *Contiki* kernel. Furthermore, the example highlights the portability of *Contiki* applications as it only uses APIs provided by the operating system, such as the event timer system. The application developer does not need to take care how waiting of one second is realized on a specific hardware platform.

3.3 *Contiki* Kernel Scheduling Mechanism

The *Contiki* system is composed of events and processes, where events are used to invoke processes of the system. An event in Contiki consists of an event type, a data part, and a target process to which an event is sent. The events are stored in an event queue. Processes in *Contiki* are applications or system tasks, whereby only one process is actively running on the processor of the system at a time. *Contiki* uses non-preemptive multitasking, meaning that processes have to give up control them self (process yields) to allow other processes to run[5]. The main task of the Contiki scheduler (or main loop) is to take an event from the event queue and invoke the related process.

[5] An optional library which supports preemptive multitasking is available for Contiki. Systems and application using this library are not discussed in this work.

To describe the event-driven kernel and the scheduler in the following, a more formal notation is introduced.

Definition 3.1. An ***event*** is a tuple $E = \langle e_{type}, e_{process}, e_{data} \rangle$ where

- e_{type} is a string defining the type of the event.
- $e_{process}$ is a pointer to the target process structure P to be called. When all processes shall be called the value points to *null* ($ProcessBroadcast$).
- e_{data} is a pointer to a memory location containing arbitrary data.

Contiki applications are allowed to create own event types by giving them a custom name. A summary of all event types that are predefined in the kernel can be found in Appendix A.2, examples are $ProcessInit$ and $ProcessEventPoll$.

In *Contiki* events can be used in two different ways:

- *Synchronous events* suspend the current running process and call immediately the target process. Afterwards the calling process resumes execution.
- *Asynchronous events*, however, do not suspend the execution of the running process. When an event is posted asynchronously it gets stored in an *event queue* for execution in the future.

Definition 3.2. Let $\mathcal{E}^{MaxEvents}$ be a *queue* of events with $\mathcal{E}^{MaxEvents} = [e_0, .., e_n]$ with $n < MaxEvents$ and $n \in \mathbb{N}$ and e being an arbitrary event. Such a queue is called ***event queue***.

By controlling $MaxEvents$ the memory allocation for the maximum number of events can be controlled. This allows it to adapt the queue size for different application configurations and hardware platforms.

To handle the execution of processes several control structures are used by the kernel to store all information related to invoking a process and to store its state when it yields.

Definition 3.3. A ***local continuation*** p_{cont} is used to store the return point of a process.

A *local continuation* is written in *Contiki* when a process yields and used to invoke a process[6].

Definition 3.4. A ***Contiki process*** is a non-preemtable entity of code.

[6]A description of the implementation of local continuations is shown Section 3.4.

Such a *Contiki* process is invoked using p_{cont}, e_{type}, and e_{data}.

Definition 3.5. A ***process structure*** is a tuple $P = \langle p_{func}, p_{cont}, p_{polled}, p_{state} \rangle$ with:

- p_{func} is a pointer to the function implementing the actual behavior of the *Contiki* process.
- p_{cont} is a local continuation storing the return value of a process,
- $p_{polled} \in \{true, false\}$ is a flag indicating, whether this process needs to be polled,
- $p_{state} \in \{None, Running, Called\}$ indicates the state of the process.

Definition 3.6. Let $\mathcal{P}$ be an *list* of process structures currently running on the system. Such a list is called ***process list***.

From an implementation point of view, each process structure has a pointer to the next process structure so that the process list is implemented as a linked list. Whenever a process needs to be used, it is iterated over this list of process structures using a loop.

```
    Variable declarations:
 1  𝒫 ← ∅;                                                        /* List of processes */
 2  ℰ^MaxEvents ← ∅;                                              /* Queue of events */
 3  E ← ∅ ;                                  /* Variable for storing an arbitrary event */
 4  PollRequested ← false ; /* Flags a poll-request, can be set to true by interrupts */
 5  System startup phase...;
 6  while (true) do
 7      DoPoll();
 8      while ℰ^MaxEvents.size > 0 do
 9          E ← ℰ^MaxEvents.dequeue;                   /* Removes first event from queue */
10          if E.e_process = ProcessBroadcast then /* Event is sent to all processes */
11              foreach P in 𝒫 do
12                  DoPoll();                   /* Do a poll between calling of processes */
13                  CallProcess(E.e_type,P,E.e_data);
14          else                                       /* Event is sent to specific process */
15              if E.e_type = ProcessInit then
16                  E.e_process ↦ p_state ← Running;
17              CallProcess(E.e_type,E.e_process, E.e_data);
18      Optional system sleep phase;
```

Algorithm 3.1: *Contiki* kernel initialization and main loop.

`DoPoll()`

Globals used: $PollRequested$, $\mathcal{P}$

1. **if** $PollRequested = true$ **then**
2. $\quad PollRequested \leftarrow false$;
3. $\quad$ **foreach** P *in* $\mathcal{P}$ **do**
4. $\quad\quad$ **if** $P.p_{polled} = true$ **then** /* check process for poll-request */
5. $\quad\quad\quad P.p_{state} \leftarrow Running$;
6. $\quad\quad\quad P.p_{polled} \leftarrow false$;
7. $\quad\quad\quad$ CallProcess($ProcessEventPoll, P, null$);

Algorithm 3.2: Function `DoPoll`: Handling of poll-requests.

`CallProcess(`e_{type}, $e_{process}$, e_{data}`)`

Input: e_{type}, $e_{process}$, e_{data}

Variable declarations:

1. $ProcessReturn \leftarrow \emptyset$; /* Return value of process execution */
2. **if** $e_{process} \mapsto p_{state} = Running$ **then**
3. $\quad e_{process} \mapsto p_{state} \leftarrow Called$;
4. $\quad ProcessReturn \leftarrow e_{process} \mapsto p_{func}(e_{process} \mapsto p_{cont}, e_{type}, e_{data})$;
5. $\quad$ **if** $(e_{type}{=}ProcessEventExit) \vee (ProcessReturn = PtEnded) \vee (ProcessReturn = PtExited)$ **then**
6. $\quad\quad$ ExitProcess($e_{process}$)
7. $\quad$ **else**
8. $\quad\quad e_{process} \mapsto p_{state} \leftarrow Running$;

Algorithm 3.3: Function `CallProcess`: Invokes a process using an event.

`ExitProcess(`P`)`

Input: pointer P to a process structure

Globals used: $\mathcal{P}$

1. $P \mapsto p_{state} \leftarrow None$;
2. **foreach** Q *in* $\mathcal{P}$ **do**
3. $\quad$ **if** $(*P) \neq Q$ **then** /* Send exit event to other processes */
4. $\quad\quad$ CallProcess($ProcessEventExited, Q, P$);
5. $\mathcal{P}.remove$(*P);

Algorithm 3.4: Function `ExitProcess`: Exits a process and removes it from the process list.

3.3.1 Description of the Scheduling Algorithm

The detailed scheduling algorithm of *Contiki* is shown in the pseudo code in Algorithm 3.1. The functions called in the main algorithm are shown in Algorithms 3.2 - 3.4. First an overview of the main scheduling loop is given, afterwards, the used functions are explained.

A non-formal description of the basic scheduling principles of *Contiki* can be found in [DGV04] and [DSVA06]. The presented algorithm in this thesis has been extracted from the *Contiki* **C** source code. Further implementation details of the *Contiki* kernel can be found in the *Contiki* documentation and source code available at [Con17a].

***Contiki* Kernel Initialization and Main Loop** Algorithm 3.1 shows the main loop of the scheduler and the initialization of the operating system. In lines 1 to 4 the global variables that are used in the algorithm such as $\mathcal{E}^{MaxEvents}$ and $\mathcal{P}$ are declared. The variable E declared in line 3 is used to store the event which is currently being processed. The variable *PollRequested* declared in line 4 is used globally to signal that, for an arbitrary process, a poll has been requested.

Before the actual processing of events in the main loop of the scheduler starts the system has to be initialized (line 5). In this phase, hardware dependent registers are set and initialization functions are called. In addition, all processes which shall be run when the system starts (`AUTOSTART_PROCESSES` in Figure 3.2) are initialized by posting a synchronous event ($ProcessInit$) to these processes.

The main loop of the *Contiki* operating system is shown in lines 6-18. This loop is an unbounded loop checking whether new (asynchronous) events or poll-requests are waiting to be processed. In line 7 it is checked using the function `DoPoll`, whether a pending poll-request needs to be handled. A poll is usually requested in an interrupt handler function, to trigger the activation of a process. By just requesting a poll, interrupts do not need to post events, avoiding race conditions in the kernel. Polls are always handled before a new event is processed and therefore have a higher priority than events.

After the poll has been processed it is checked, whether an event is in the event queue waiting to be processed (line 8). If it is available, it gets removed and stored in the temporary variable E (line 9). When the event is of type $ProcessBroadcast$, it has to be sent to all processes (line 10). In between the calling of all the processes in the process list, the `DoPoll` function is called (line 12). This ensures that processes, which have been requested to be polled by an interrupt in between, get handled with higher priority. The actual calling of a process is handled using the `CallProcess` (line 13) function, which takes as argument the information provided by an event. In the case

of a *ProcessBroadcast*, the target processes are all processes in $\mathcal{P}$.

If the event is being sent to a specific process (else branch, line 14) it is first checked, whether the event is of type *ProcessInit*[7] (line 15). In this case, the state of the process called by the event is changed in the process structure to *Running*. Afterwards the process gets called using the `CallProcess` function (line 17).

After events have been processed, a hardware specific low power mode is often enabled (line 18) when $\mathcal{E}^{MaxEvents}$ is empty. This allows it to save power when no events need to be processed. Such a low-power mode is usually exited, when an interrupt occurs. Such an interrupt causes the system to continue and often leads to several events which need to be processed.

Function `DoPoll` (Algorithm 2) This function handles the poll-requests, which are usually triggered by interrupt routines. The function works on the global variables *PollRequested* and the process list $\mathcal{P}$, it is not called with any function parameters.

At the beginning it is checked whether any poll has been requested at all that needs to be handled (line 1). When such a request exists, the variable *PollRequested* is set to false to signal that polls have been handled (line 2). Afterwards, for each process in $\mathcal{P}$, it is checked whether this process has requested a poll (line 4). If this is true then the state of the process is set to *Running* (line 5) and the flag indicating that a process needs to be polled is reset (line 6). Then the process which needs to be polled is called using the special event type *ProcessEventPoll* (line 7) and without any data associated.

Function `CallProcess` (Algorithm 3) This function calls the actual process associated with a process structure using an event. This function takes as an argument an event type e_{type}, a pointer to a process structure $e_{process}$, and data sent with an event e_{data}. A local variable *ProcessReturn* is declared to store the return value of the process execution (line 1).

In the function, it is first made sure that only a process is invoked, which is actually running (line 2). After the state of the process is changed to *Called* (line 3). The actual function which contains the code of the process is executed using the arguments needed to invoke a process. The return value of this process function is stored in the variable (line 4). Depending on the state of this variable, it is checked in line 5 whether the process has finished. When this is the case the function `ExitProcess` is called (line 6). A process is also always exited when it receives the event *ProcessEventExit*. When the process is yielding and has not finished, the state of the process is changed back

[7] A list of predefined *Contiki* events can be found in Appendix A.2.1.

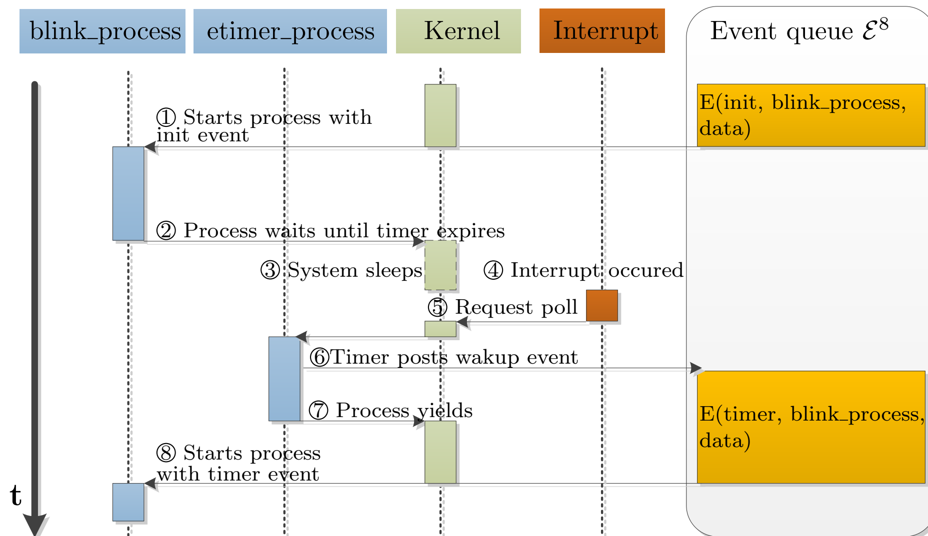


Figure 3.3: Contiki process communication in the *LED Blink* example.

to running (line 8).

Function `ExitProcess` (Algorithm 4) This function describes how a process structure is removed from $\mathcal{P}$. It works on $\mathcal{P}$ and takes as an argument the process structure P which controls the process that shall be exited.

At the beginning, the state of the process is set to *None* (line 1). Afterwards, all other running processes in the system are informed that process P is about to exit (lines 2-4). As a parameter, the special event $ProcessEventExit$ is used as well as the pointer to the process structure P as data part (line 4). This call to other processes allows them to deallocate memory associated with P. Finally, the process structure P gets removed from the process list $\mathcal{P}$ (line 5).

3.3.2 Scheduling of the *LED Blink* Example

In Figure 3.3 the process communication for the *LED Blink* example introduced in Section 3.2 is shown exemplarily when the application would run on a processor that uses interrupts to control the event timer system such as the MSP430. It shows how the involved processes of the example are executed sequentially. An event queue $\mathcal{E}^8$ with the size restriction of 8 is used, which is the default size for the hardware platform the application is executed on. The figure corresponds to lines 1-10 of the *LED Blink* application in Figure 3.2.

First, to launch the application process, an init event is used to start the application. Therefore, the *Contiki* scheduler starts the blink_process using the process initializa-

tion event *ProcessInit* ①. After initializing the event timer system the blink_process ② yields control to the kernel using the `PROCESS_WAIT_EVENT_UNTIL` macro. When no other process is scheduled, the system goes to a sleep mode ③ until a timer interrupt wakes the system up ④. In the corresponding timer interrupt function the system time is incremented, and it is checked whether an event timer has expired. When this is the case, using the poll mechanism, the etimer_process is scheduled for immediate execution ⑤. In the etimer_process it is checked, which processes have pending timer requests and wait to be resumed. For each waiting timer request, an asynchronous event is posted to the event queue to wake it up ⑥. Afterwards, the etimer_process yields control back to the kernel ⑦, which then posts the timer event to the blink_process ⑧. As the timer condition in line 10 of the *LED Blink* application is now fulfilled, it will resume execution and turn on the LED in line 11.

3.4 Programming *Contiki* Applications

An event-driven kernel as used in Contiki or TinyOS has the advantage of a low memory overhead and fast reaction times, as no stack is saved when an application context changes. Furthermore, as no preemption is used, processes have to give up control themselves to not block the system [LMP+05]. Therefore, programming applications for event-driven operating systems is challenging, as the programmer has to manually store the state of the application and often use complex state machines. To make programs easier understandable, *Contiki* applications are written using the *protothread* programming model introduced in [DSVA06]. This programming model allows it to write event-driven applications in a process-like fashion, without the need to explicitly store the state of a process when it yields control to the kernel. The introduction of *protothreads* leads to a slight overhead compared to a pure event-driven kernel, however, this overhead is still very low as shown in [DSVA06].

The different `PROCESS_*` macros introduced exemplarily in Section 3.2 are part of the *protothread* programming model and are based on the principle of *local continuations* which are used to store the return point of a protothread. A `LC_SET` macro stores the return point of a process whereas `LC_RESUME` is used to restore the point at which a process was suspended.

The default[8] *protothread* implementation available in *Contiki* is written completely in **C** and is therefore hardware and compiler independent. The implementation is relying on the preprocessor of the used **C** compiler to store the line number of the statement where a process should return to, as a kind of label. A `switch`-statement is then used

[8] Different implementations for processors that support the GCC compiler [GCC17], as well as an assembler version for the 6502 architecture [Mos15], are also available.

to return to this label. **C** allows it to place the `case` part of `switch` nearly everywhere in the code even in separate loops. This programming style has been widely used [Sim00].

In Figure 3.4 it is shown exemplarily how the *protothread* macros `PROCESS_BEGIN` and `PROCESS_WAIT_EVENT_UNTIL` used in Figure 3.2 are expanded and implemented. The `PROCESS_*` macros are mapped on a subset of *protothread* macros called `PT_*`. On the left-hand side, it is shown that the `PROCESS_BEGIN` macro defines the `switch`-statement that is used to jump to the return point of a process based on the `LC_RESUME` macro. On the right-hand side, it can be seen how a return point is set. The `LC_SET` macro is extended so that it stores the line number of the return point (55) in the local continuation of the process. This line number is then used as a label for the `case` part of the `switch` statement. The `PROCESS_WAIT_EVENT_UNTIL` macro always yields and only continues when the associated expression (`etimer_expired(&et)`) is true. This is checked in the `if`-statement part of the macro. A more complex example of using *protothreads* showing the use of the `PT_*` macros is the *Contiki* fader example in Section 6.4 with the source codes shown in Appendix A.1.4. An overview of the available process macros in *Contiki* is given in A.2.

The default implementation of the *protothread* model leads to some restrictions:

- In [DSVA06] it is warned not to use `switch`-statements in *protothreads*, as nested `switch`-statements can lead to unintended behavior.
- Only one `PROCESS_*` macro should be used per line as the label for the `case` is based on the line number of the code.
- As the stack is not saved when yielding, static variables have to be used when storing values that need to be restored after a *protothread* yields.

3.5 Hardware Access in *Contiki*

Contiki is designed as a portable operating system for different hardware platforms. Therefore, access to the hardware for many basic tasks is provided using predefined APIs, which allow it to write applications without exact knowledge of the target platform. Two examples of the use of such hardware APIs can be seen in the simple example application of Figure 3.2. The LED API functions `led_on` and `led_off` as well as the macro `LEDS_ALL` used in the *LED Blink* example are provided by the *Contiki* system and can be implemented by each hardware platform.

Furthermore, the event timer system used in the example demonstrates the hardware abstraction enabled by Contiki. The system allows it to suspend a process for a specific time period, which can be specified for example in seconds. During this waiting time,

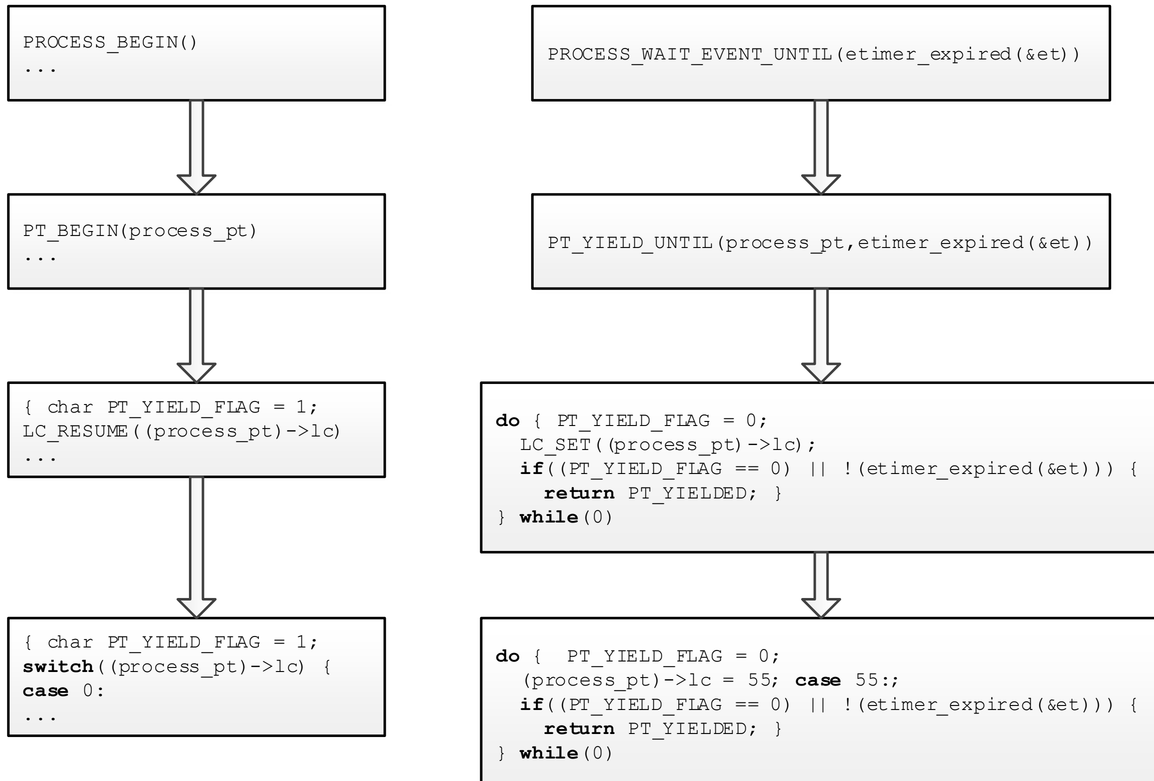


Figure 3.4: Implementation of the *protothread* macros `PROCESS_BEGIN` and `PROCESS_WAIT_EVENT_UNTIL`.

```
contiki_src  // Contiki main directory
 |__apps    // applications bundled with Contiki
 |__core    // operating system core
 |   |__...
 |   |__dev  // specifies APIs for commonly used devices
 |   |__lib  // library of predefined functions(e.g.  CRC)
 |   |__sys  // system files including event and process definitions
 |   |__...
 |__cpu    // CPU-specific code
 |   |__...
 |   |__arm
 |   |__avr
 |   |__msp430
 |   |__x86
 |   |__...
 |__doc      // Contiki doxygen documentation
 |__examples // Collection of example programs
 |__platform // platform-specific code
 |   |__...
 |   |__avr-raven
 |   |__cooja
 |   |__sky
 |   |__texnode
 |   |__win32
 |   |__...
 |__tools // additional software tools for using Contiki
```

Figure 3.5: Organization of the *Contiki* source code.

other processes can run. After the time is over the waiting process is reactivated using an event. The realization of the timing system is completely encapsulated from the application and doesn't enforce a hardware implementation. Additionally, APIs provide access to often used devices in embedded systems. These APIs include generic devices such as LEDs or sensors and can be implemented for a specific embedded system platform allowing applications to be portable. In case no appropriate API exists, *Contiki* can be easily extended.

To increase the portability and ease the creation of new sensor node boards, the *Contiki* source code implementation divides the *hardware dependent* parts of the operating system code into *CPU-specific* and *platform-specific* code. A general overview of the structure of the *Contiki* source code is given in Figure 3.5. The separation between CPU and platform is made, as many embedded system platforms use the same kind of CPU but use different kinds of peripheral hardware:

- The *CPU-specific* code contains all code which is necessary to run *Contiki* on a processor. These are mostly interrupt handlers needed for timers running on the CPU, access functions to hardware interfaces often used in microcontrollers such as SPI, UART or I^2C, as well as functions to enable the low-power mode of a CPU.
- The *platform-specific* code encapsulates all code needed to access peripheral hardware not part of the CPU including sensors, displays or external networking. The function, which implements the LED API used in the *LED Blink* example is implemented as part of the platform-specific code and maps the LED API functions to the hardware. In addition, the platform-specific code also contains the **C** *main*-function used to start the *Contiki* system and the configuration of the embedded system. This configuration includes hardware parameters like the clock frequency of the CPU and *Contiki*-specific options as the size of the *Contiki* event queue and the processes to be started during system startup.

Therefore, when adapting *Contiki* to a newly built hardware platform that uses an already supported processor, it is often only necessary to create a new platform-specific code section, which configures the system and implements the used drivers. This allows it to reuse applications between different embedded system platforms[9].

[9] A recompilation with the CPU specific compilers is necessary.

Chapter 4

4 Modeling an Embedded System Running Contiki for Verification

In the *modeling phase* of the model checking process (cf. Section 2.2.1) the system description and properties, both derived from the system specification, have to be created. This chapter describes how the components of an embedded system are modeled to allow the verification using a software model checking tool. Furthermore, assertions, which describe the properties to be checked, are classified.

The chapter is structured as follows: In Section 4.1 an overview of the general verification approach is given. The representation of the system specification by the use of assertions is described in Section 4.2. The general approach for modeling drivers of a *Contiki* system is given in Section 4.3. A special task when modeling a *Contiki* system is the modeling of interrupts. Existing approaches for interrupt modeling are discussed in Section 4.4. In Section 4.5 a new approach called *periodic interrupt modeling* suited for the verification of software using periodic interrupts is presented.

4.1 Overview of the Approach

The aim of the presented verification approach is to allow the verification of *Contiki* applications without modifications of the application source code[10]. As the behavior of a *Contiki* application strongly depends on the surrounding hardware of an embedded systems, also this environment must be modeled.

[10]The general restrictions for software model checking described in Section 2.2.4 however still apply. Especially the complete source code must be available and only **C** libraries supported by CBMC can be used.

The general approach is depicted in Figure 4.1. Figure 4.1(a) shows how a *Contiki* application normally interacts with the hardware of an embedded system. An application can access the hardware through the *Contiki* predefined driver APIs. In the same way, also the processes, which are part of the operating system, interact with the hardware using these APIs, e.g. the timer subsystem. Moreover, applications can interact with the hardware using platform-specific drivers, for which *Contiki* does not provide an API. To perform the verification, the actual drivers, which interact with the system hardware are replaced with verification models, which capture the hardware behavior as shown in Figure 4.1(b). In addition, the drivers and the application to be verified can be annotated with assertions, which must hold on the system. The practical challenge when creating abstract drivers is to model the hardware behavior at the right level of abstraction. When the driver is not modeled appropriately, either bugs can be missed or unrealistic counterexamples will be generated, which cannot occur on the real hardware.

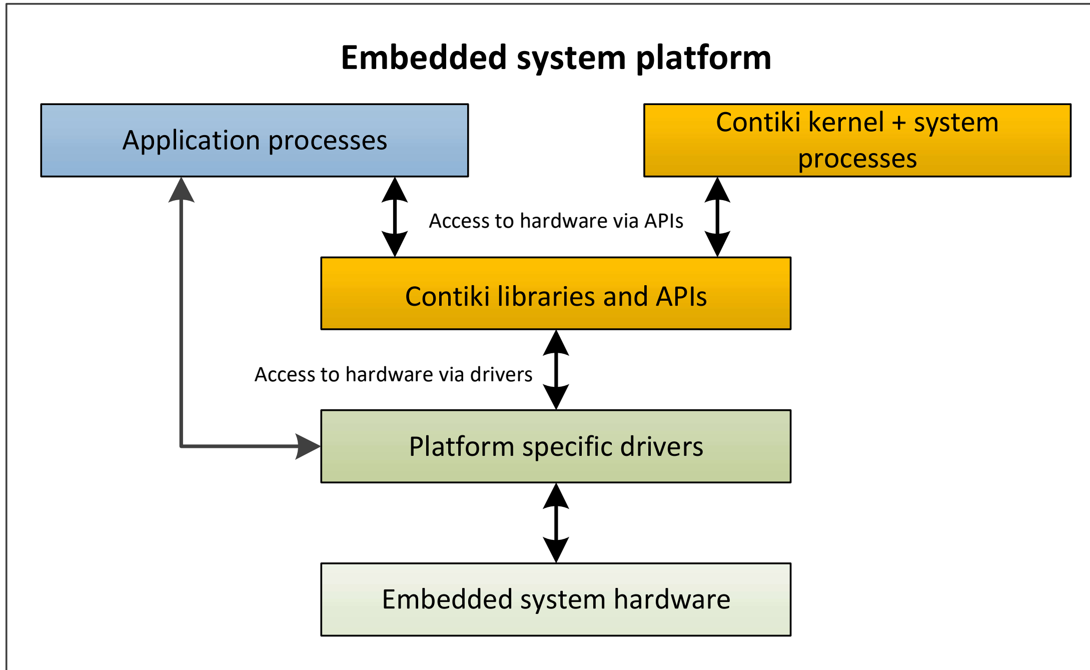
As an example for the approach, consider the *LED Blink* example shown in Figure 3.2. There, two drivers of an embedded system must be modeled: The LED API of *Contiki*, which allows the blinking of the LED and the event timer API, which realizes the waiting for several seconds.

By replacing only hardware related code, it is guaranteed that no modeling errors occur within the application software, as it stays unmodified. However, a restriction of the presented approach is that the actual implementation of the driver is not verified, when modeling the hardware access at the level of a driver API. Consequently, no bugs can be detected, which are caused by the driver implementation e.g. an arithmetic overflow in a driver.

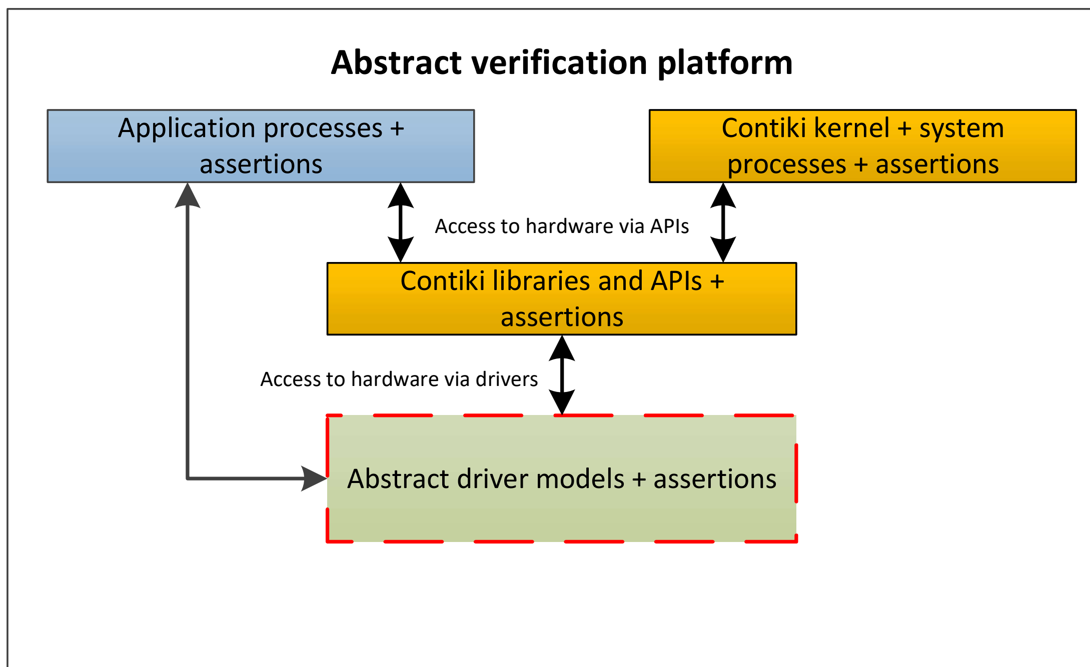
In this thesis, for modeling the CBMC tool is used and therefore all modeling, also of the embedded system hardware is done using the **C** language. To represent the non-deterministic behavior and input ranges, constructs provided by CBMC are used in the following examples. Although this thesis builds on CBMC, the concepts of the modeling approach can be also applied with other model checking tools too, as they provide similar modeling constructs.

4.2 Annotating Assertions for Verification

The input for the model checker shall be the *Contiki* source code including applications and hardware models as overall system model. In addition, assertions have to be provided as system specification. Using these assertions it can be checked, whether the software accesses the hardware correctly, uses the operating system correctly, and



(a) Embedded system platform running *Contiki* applications.



(b) Abstract verification platform used for verification.

Figure 4.1: Replacement of drivers and annotation of assertions for an abstract *Contiki* verification platform.

whether it is algorithmically correct. Therefore, the assertions[11] can be grouped depending on the location within the *Contiki* system:

- Application-dependent assertions
- *Contiki*-specific assertions
- Platform-dependent assertions
- Automatically generated assertions

A description of these assertions follows.

Application-dependent assertions check whether the application performs algorithmically correct. They are added to the source code of an application running on a *Contiki* system and are derived from the application specification. An example of such an assertion is the program cutout shown in Figure 4.2(a). The function `checkSortedArray` checks that the passed array `a` with size `n` is sorted correctly by looping through the array and checking that each element is smaller or equal to its successor using the assertion in line 5.

***Contiki*-specific assertions** have to be valid for all *Contiki* applications, independently of a specific embedded system platform. They check, for example, that a *Contiki* API is accessed correctly from an application. As these assertions are hardware independent, they have to be written only once and must hold for all *Contiki* applications.

One typical assertion for *Contiki* is checking that the size of the event queue configured for a system is sufficient. When this size is not sufficient, an application cannot create new events anymore. Such a check can be added to the *Contiki* kernel in the `process_post` function depicted in Figure 4.2(b). This function is used to post asynchronous events (see Section 3.3) when invoking processes. These events are stored in the event queue till they are processed. The function already prints warnings in a *Contiki* debug mode, to identify problems in the case variable `nevents` - used to count the current size of the event queue - reaches the maximum size defined for the event queue (lines 3-9 of Figure 4.2(b)). Similarly, an `assert(0)` (line 12) statement can be added, denoting a piece of code which must not be reached as the assertion will always be violated, when the code is reachable. The assertion is guarded in this case by a macro and can be activated when the code is compiled for verification (cf. Section 5.2.2).

[11] CBMC supports only safety properties via the `assert` statement as described in Section 2.2.4.

Platform-dependent assertions are used to check, whether processes correctly access the drivers of a hardware component. These assertions are defined once for each embedded system platform hardware and must hold for all applications running on that platform. They make sure that a hardware API is used correctly. Figure 4.2(c) shows a cutout of a platform-dependent driver for an LC display, whereby the shown function prints a symbol at the specified position of the display. The device can only show a restricted number of symbols and has to be initialized before it can be used. The three assertions check that the initialization has occurred before the function is called (line 2), the display position of the symbol is valid (line 3), and only a valid symbol is chosen to be printed (line 4).

Automatically generated assertions can be added by the CBMC tool as described in Section 2.2.4. These assertions include checks for array bounds, arithmetic overflows, division by zero, and the absence of null pointer dereferences in the code. For example, the CBMC tool can add automatic checks for array bounds, whenever an array is accessed, as in the function shown in Figure 4.2(a). As the size of arrays in **C** cannot be deduced from the array itself, this size must be known and is therefore passed as an additional parameter. If the array size is assumed wrong, an out-of-bound array access can occur. Therefore, CBMC can automatically add assertions, which check that, for example, the access to the array in line 5 via the used indexes `i` and `i+1` is never out of bounds.

The *application-dependent assertions* are optional and need to be defined by the author of the application. When they are omitted, it is only checked whether the application fulfills *platform-dependent assertions*, *Contiki-specific assertions*, and *automatically generated assertions* (when activated). Using this approach it is possible to check completely unmodified *Contiki* applications. Automatically generated assertions such as for array bounds are very useful from a practical point of view, as these detect common programming errors, which can lead to a runtime failure of the software, crashing the complete system. Examples of all kinds of assertions will be shown in the next sections and in the examples in Chapter 6.

4.3 Modeling the System Environment Using Drivers

To represent the system environment of an embedded system, all possible behaviors of the hardware must be resembled by a model, with which the software can interact using APIs. This model must also be annotated with assertions to make sure that the

```
1 void checkSortedArray(uint8_t a[], uint8_t n)
2 {
3   uint8_t i;
4   for (i=0; i<n−1; ){
5     assert (a[i] <= a[i+1]);
6     i=i+1;
7   }
8 }
```

(a) Assertion checking that an array is sorted.

```
1  if(nevents == PROCESS_CONF_NUMEVENTS) {
2  #if DEBUG
3    if(p == PROCESS_BROADCAST) {
4      printf("soft panic: event queue is full when broadcast event %d was posted from %s\n",
5      ev, PROCESS_NAME_STRING(process_current));
6    } else {
7      printf("soft panic: event queue is full when event %d was posted to %s from %s\n",
8      ev, PROCESS_NAME_STRING(p), PROCESS_NAME_STRING(process_current));
9    }
10 #endif
11 #ifdef __ASSERTION_CHECK_EVENT_QUEUE_OVERFLOW
12   assert(0);
13 #endif
14   return PROCESS_ERR_FULL;
15 }
```

(b) Assertion checking that an event queue overflow must not occur; part of the `process_post` function of the *Contiki* kernel.

```
1 void lcd_disp_char(uint8_t pos, uint8_t index) {
2   assert (init == 1);
3   assert(pos < LCD_NUM_DIGITS);
4   assert(index < LCD_MAX_CHARS);
5 }
```

(c) Assertions checking that an LCD driver is used correctly.

Figure 4.2: Example assertions for the verification of *Contiki* based systems

API is correctly used by the application. In this section, the modeling of drivers, which are not based on interrupts is shown, whereas in Section 4.4 the use of interrupts is described in detail.

To demonstrate the approach cutouts representing three typical kinds of drivers for the *Contiki* system are shown.

- The first driver demonstrates a hardware device, which does not give any feedback to the system such as an LED. This kind of driver can therefore be easily integrated.
- The second driver provides undetermined feedback, which is not under control of the software and modeled using non-deterministic variables. The chosen example is taken from a memory card driver.
- With the third driver, the use of *platform-dependent assertions* on a more complex driver is demonstrated. This driver is based on a 3-axis acceleration sensor used, e.g., in fall detection applications.

First, the *Contiki* LED API as used in the *LED Blink* example in Section 3.2 is considered. The API provides functions such as `leds_on` or `leds_off` for turning an LED on or off, as shown in Figure 4.3(a). For implementing the API a hardware platform must provide the functions `leds_arch_init` (initialization of the LEDs), `leds_arch_get` (returns the current LED status), and `leds_arch_set` (turns LEDs on/off). Using this API up to 8 LEDs can be controlled using a `char` value, where each bit passed to the `leds_arch_set` function represents one LED.

The driver implementation provided by *Contiki* for an MSP430 microcontroller is shown in Figure 4.3(b). This driver implementation supports three LEDs (red, green, yellow) and is configurable, depending on which hardware ports of the microcontroller the LEDs are connected to, using the macro `LEDS_PxOUT`. This port, as well as the pin to which the LED of the corresponding color is connected (macros `LEDS_CONF_RED`, `LEDS_CONF_YELLOW`, `LEDS_CONF_GREEN`), must be set for the configuration of the driver to work together with the hardware platform. The verification model of the driver is shown in Figure 4.3(c) and stores the value of the set LEDs in a static variable, so that it can be read back using the `leds_arch_get` function[12]. In addition, this driver shows how assertions can be added. Similar to the LCD driver described in Section 4.2 (cf. Figure 4.2(c)), a global variable `led_init` is used, which makes sure that the function `leds_arch_init` is called before other functions of the LED API are called.

The second class of drivers is non-deterministic - based on the environment of the system, which is not controlled by the software - and can provide random feedback. A

[12] The used `printf` function is built into CBMC and is evaluated during counterexample generation, making it easier to debug the generated traces.

```
1  /**
2   * Returns the current status of all leds (respects invert)
3   */
4  unsigned char leds_get(void);
5  void leds_on(unsigned char leds);
6  void leds_off(unsigned char leds);
7  void leds_toggle(unsigned char leds);
8  void leds_invert(unsigned char leds);
9
10 /**
11  * Leds implementation
12  */
13 void leds_arch_init(void);
14 unsigned char leds_arch_get(void);
15 void leds_arch_set(unsigned char leds);
16
```

(a) LED API including platform-specific functions to be implemented.

```
1  void leds_arch_init(void)
2  {
3    LEDS_PxDIR |= (LEDS_CONF_RED | LEDS_CONF_GREEN | LEDS_CONF_YELLOW);
4    LEDS_PxOUT |= (LEDS_CONF_RED | LEDS_CONF_GREEN | LEDS_CONF_YELLOW);
5  }
6  unsigned char leds_arch_get(void)
7  {
8    return ((LEDS_PxOUT & LEDS_CONF_RED) ? 0 : LEDS_RED)
9      | ((LEDS_PxOUT & LEDS_CONF_GREEN) ? 0 : LEDS_GREEN)
10     | ((LEDS_PxOUT & LEDS_CONF_YELLOW) ? 0 : LEDS_YELLOW);
11 }
12 void leds_arch_set(unsigned char leds)
13 {
14   LEDS_PxOUT = (LEDS_PxOUT
15     & ~(LEDS_CONF_RED|LEDS_CONF_GREEN|LEDS_CONF_YELLOW))
16     | ((leds & LEDS_RED) ? 0 : LEDS_CONF_RED)
17     | ((leds & LEDS_GREEN) ? 0 : LEDS_CONF_GREEN)
18     | ((leds & LEDS_YELLOW) ? 0 : LEDS_CONF_YELLOW);
19 }
```

(b) Implementation of the LED API for an MSP430 microcontroller.

Figure 4.3: *Contiki* LED API and implementations for hardware access and verification.

```
1  static unsigned char leds;
2  void leds_arch_init(void)
3  {
4    printf("LED Driver: Initializing LED \n");
5    led_init = 1;
6    leds = 0;
7  }
8  unsigned char leds_arch_get(void)
9  {
10   assert (led_init == 1);
11   return leds;
12 }
13 void leds_arch_set(unsigned char l)
14 {
15   assert (led_init == 1);
16   printf("LED Driver: Setting led to %d\n", l);
17   leds = l;
18 }
```

(c) Implementation of the LED API for formal verification.

Figure 4.3: *Contiki* LED API and platform implementations for hardware access and verification.

```
1  char mmc_ping(void) {
2  if (!(MMC_CD_PxIN & MMC_CD))
3   return (MMC_SUCCESS);
4  else
5   return (MMC_INIT_ERROR);}
```

(a) Driver for hardware access.

```
1  char mmc_ping(void) {
2  char rvalue = nondet_char();
3  __CPROVER_assume(rvalue == MMC_SUCCESS
4  || rvalue == MMC_INIT_ERROR);
5  return rvalue; }
```

(b) Model for formal verification.

Figure 4.4: Cutout from memory card driver as used in *Contiki*.

function, belonging to a memory card driver is shown in Figure 4.4, which - depending on whether a memory card is available or not - returns either the values `MMC_SUCCESS` or `MMC_INIT_ERROR`. In Figure 4.4(a) the original hardware driver is shown, which communicates with the hardware using the pins `MMC_CD_PxIN` and `MMC_CD`. The corresponding model for verification is shown in Figure 4.4(b). To represent that either a memory card can be present or missing, non-deterministic return values are used, which are restricted using the statement `__CPROVER_assume`. When using this driver the state space of an application is searched for all possibilities, with the function returning either the value `MMC_SUCCESS` or `MMC_INIT_ERROR`.

An even more complex model of a driver is the model of a 3-axis acceleration sensor. A cutout from this driver is given in Figure 4.5. This sensor has first to be initialized (function `adxl345_init_dev`) before it can be used, and measurement has to be enabled (function `adxl345_enable_measurement`) so that sensor values can be retrieved (function `adxl345_get_xyz`). The original driver (shown in Figure 4.5(a)) uses an I^2C bus to communicate with the sensor, which can – in this example – transmit several 8-bits fields using the `i2c_data_tx` and `i2c_data_rx` functions. As the sensor provides 16-bit values for the acceleration for each axis, 6 values must be read from the sensor. Therefore, to read out the sensor values a pointer to an array of bit values is passed to the `adxl345_get_xyz` function (line 13).

For the abstract model of the driver, which is shown in Figure 4.5(b), the return values of the sensor must be provided. In addition, also assertions were added, which check the correct usage of the driver by the software. Therefore, the variable `init` stores the state of the sensor. When it has value 0 the sensor is not initialized. Value 1 signals, that the driver is initialized, but no measurement has yet been enabled and value 2 indicates that the measurement is enabled. In the functions `adxl345_enable_measurement` and `adxl345_get_xyz`, it is checked whether the variable has the correct state each time these functions are called. The return values of the sensor are not restricted and, thus, modeled using non-deterministic variables (lines 14-16).

Discussion of the approach The shown modeling approach abstracts the hardware at the level of the API provided by the driver or the operating system. Instead of detailed hardware models as used in hardware description languages such as SystemC, return values are described using non-deterministic values. The detail level of the models could, of course, be increased using a finite state machine, which can store the state and therefore the history of driver calls. Furthermore, it is possible to access the global system time variable of *Contiki* to describe even more complex behavior. However, it has to be taken into account that the detail level of the model directly

```
1  uint8_t xyz_data[6] = {0,0,0,0,0,0};
2  void adxl345_init_dev() {
3          i2c_init_dev();
4          char byte[1] = {0x09}; //set data rat to 50Hz
5          i2c_data_tx(ACC_ADDR, 0x2c, ACC_ADDR_SIZE, byte, 1);
6          byte[0] = 0x08; //set data format register (full resolution, +/-2g range)
7          i2c_data_tx(ACC_ADDR, 0x31, ACC_ADDR_SIZE, byte, 1);
8  }
9  void adxl345_enable_measurement() {
10         char byte[1] = {0x08};
11         i2c_data_tx(ACC_ADDR, 0x2d, ACC_ADDR_SIZE, byte, 1);
12 }
13 void adxl345_get_xyz(int16_t *xyz) {
14   i2c_data_rx(ACC_ADDR, 0x32, ACC_ADDR_SIZE, xyz_data, 6);
15   *xyz = (xyz_data[1]<<8 | xyz_data[0]);
16   *(xyz+1) = (xyz_data[3]<<8 | xyz_data[2]);
17   *(xyz+2) = (xyz_data[5]<<8 | xyz_data[4]);
18 }
```

(a) Driver for hardware access.

```
1  static uint8_t init;
2  void adxl345_init_dev() {
3    printf("ADXL Driver: Initializing Acceleration Sensor \n");
4    init = 1;
5  }
6  void adxl345_enable_measurement() {
7    assert(init == 1);
8    printf("ADXL Driver: Enabling measurement \n");
9    init = 2;
10 }
11 void adxl345_get_xyz(int16_t *xyz) {
12   printf("ADXL Driver: Requesting xyz %d \n", *xyz);
13   assert(init == 2);
14   *xyz = nondet_int16_t();
15   *(xyz+1) = nondet_int16_t();
16   *(xyz+2) = nondet_int16_t();
17 }
```

(b) Model for formal verification.

Figure 4.5: Cutout from a 3-axis acceleration sensor driver as used in *Contiki*.

impacts verification performance, as it influences the size of the state space.

Other approaches for verification of embedded systems such as [LFCJ09] or [BK11b] do not take the operating system into account and model the hardware at the level of the pins and registers of a microcontroller (particularly an MSP430 microcontroller), making it basically impossible to model external components such as sensors. For example, the 3-axis acceleration sensor mentioned above is accessed either via an SPI or an I^2C interface, and could only be modeled on the level of hardware registers for the interface provided by the microcontroller. Consequently, assumptions can only be specified on the state of these registers without taking a connected device into account.

In contrast, the approach presented in this thesis uses an abstract driver for verification with *Contiki*, which is independent of the actual interface the driver is accessed by and therefore, also independent of the microcontroller the application is running on. However, a verification of drivers and low-level code as feasible by the above-described approach is not possible by the presented method for *Contiki*.

4.4 Interrupt Modeling

Up till now, only drivers were described, in which the application initiates the hardware access by requesting the driver to deliver data to the application. An important role in the hardware of embedded systems play, however, ***interrupts***. Interrupts are another approach to let software communicate with hardware or to perform periodic tasks. Especially in low power systems, interrupts are very important, as they enable it to wake up the system when further processing needs to be done, e.g. from a sleep mode, and avoid the usage of so called *busy waiting*.

An interrupt can either be triggered periodically from a special hardware unit in the microcontroller of the embedded system, or from an external source, e.g. through designated hardware pins. When an interrupt occurs, the system is woken up from a sleep mode or - when the system is executing code - the current state of the running program is stored by the hardware. Afterwards, an interrupt service routine (ISR) is executed. An ISR is a piece of code associated with the interrupt, whose execution is triggered by the hardware. This code is then used, e.g. to access external hardware via a driver (see Section 4.3) or trigger the resumption of the system operation after a sleep mode.

In *Contiki*-based embedded systems one application of interrupts is the event timer system as described for the *LED Blink* example in Section 3.3.2. Interrupts allow the system to save power when it is not used. In the example, the use of the *protothread* macro `PROCESS_WAIT_EVENT_UNTIL` leads to a system sleep state, which is only left

when the interrupt signals that the waiting time is over. The implementation of the ISR, which calls the *Contiki* timer system is shown in Figure 4.6. Figure 4.6(a) shows the implementation available in *Contiki* for an MSP430 processor and Figure 4.6(b) the hardware independent version, which is used for formal verification. The ISR implementation for the MSP430 processor works with interrupt control registers and flags (`TACCR1, TACTL, MC1, TAR`) to check the configured clock period, which is set in clock cycles of the processor. Furthermore, the ISR has to control the power modes of the processor (`LPM4_EXIT`). Additionally, the ISR uses the *Contiki* functionality for estimating power consumption with the `ENERGEST` macros. The model for verification (Figure 4.6(b)) is derived from the original ISR by abstracting these hardware accesses. A short description of the main principle of this ISR for the *Contiki* event timer system follows.

The variables `count` and `seconds` are used to represent the overall system time, `count` is incremented whenever the interrupt is called (line 25 in Figure 4.6(a), line 2 in Figure 4.6(b)). The actual time passed between interrupt calls depends on the configuration of the timer system, which can be different for each application scenario of the embedded system. The `CLOCK_CONF_SECOND` macro configures the number of interrupt function calls corresponding to an actual second in the system. The `seconds` variable get increased correspondingly (lines 30-31 in Figure 4.6(a), lines 3-4 in Figure 4.6(b)). When an application has registered a timer, the ISR checks whether the registered time has been reached[13] (lines 36-37 in Figure 4.6(a), lines 5-6 in Figure 4.6(b)). If this is the case, the interrupt requests a poll for the event timer process (line 38 in Figure 4.6(a), line 7 in Figure 4.6(b), see also Section 3.3), waking up the system if it is in a low power mode.

The presented example only shows how an ISR can be abstracted for verification from a specific processor. However, in the verified program the ISR must be also called during system execution. There, the challenge is that interrupts can occur at any time, influencing the program flow. Each of these program flows must be considered when performing model checking. For example, when no interrupts are executed within *Contiki* the overall system time is never increased, halting the overall system progress. Modeling of interrupts is therefore especially important for capturing the correct invocation of processes, as otherwise spurious counterexamples could be introduced or properties cannot be proven.

In the following, current approaches to model interrupts are summarized in Section 4.4.1. Using a general example with interrupts, it is discussed how well these approaches are suited for the verification of *Contiki* applications (Section 4.4.2). Based

[13]This comparison is based on the fact, that `-1` corresponds to the largest possible value in an unsigned number representation and is therefore larger than the positive integer `MAX_TICKS`, which is defined in line 2 in Figure 4.6(a).

```
1  #define INTERVAL (RTIMER_ARCH_SECOND / CLOCK_SECOND)
2  #define MAX_TICKS (~((clock_time_t)0) / 2)
3  static volatile unsigned long seconds;
4  static volatile clock_time_t count = 0;
5  /* last_tar is used for calculating clock_fine */
6  static volatile uint16_t last_tar = 0;
7  #ifdef __IAR_SYSTEMS_ICC__
8  #pragma vector=TIMERA1_VECTOR
9  __interrupt void
10 #else
11 interrupt(TIMERA1_VECTOR)
12 #endif
13 timera1 (void) {
14   ENERGEST_ON(ENERGEST_TYPE_IRQ);
15   watchdog_start();
16   if(TAIV == 2) {
17     /* HW timer bug fix: Interrupt handler called before TR==CCR.
18      * Occurrs when timer state is toggled between STOP and CONT. */
19     while(TACTL & MC1 && TACCR1 - TAR == 1);
20     /* Make sure interrupt time is future */
21     do {
22       /* TACTL &= ~MC1;*/
23       TACCR1 += INTERVAL;
24       /* TACTL |= MC1;*/
25       ++count;
26 #if (CLOCK_CONF_SECOND & (CLOCK_CONF_SECOND - 1)) != 0
27 #error CLOCK_CONF_SECOND must be a power of two (i.e., 1, 2, 4, 8, 16, 32, 64, ...).
28 #error Change CLOCK_CONF_SECOND in contiki-conf.h.
29 #endif
30       if(count % CLOCK_CONF_SECOND == 0) {
31         ++seconds;
32         energest_flush();
33       }
34     } while((TACCR1 - TAR) > INTERVAL);
35     last_tar = TAR;
36     if(etimer_pending() &&
37        (etimer_next_expiration_time() - count - 1) > MAX_TICKS) {
38       etimer_request_poll();
39       LPM4_EXIT;
40     }
41   }
42   watchdog_stop();
43   ENERGEST_OFF(ENERGEST_TYPE_IRQ);
44 }
```

(a) Timer interrupt implemented for the MSP430 processor.

Figure 4.6: Cutout from the timer interrupt implementation of the *Contiki* event timer system.

```
1 void timer_interrupt(void) {
2   count++;
3   if ((count % CLOCK_CONF_SECOND) == 0)
4     seconds++;
5   if(etimer_pending() &&
6     (etimer_next_expiration_time() − count − 1) > MAX_TICKS) {
7       etimer_request_poll();
8   }
9 }
```

(b) Platform independent model for formal verification.

Figure 4.6: Cutout from the timer interrupt implementation of the *Contiki* event timer system.

on this discussion a new approach to the modeling of interrupts called *periodic interrupt modeling* is introduced in Section 4.5.

4.4.1 Existing Approaches

This section discusses existing approaches for the formal verification of software on systems which use interrupts.

In [BK11b] Bucur and Kwiatkowska show a complete formal verification flow for the *TinyOS* operating system using the model checker CBMC. As *TinyOS* applications are written in *nesC* (a programming language for *TinyOS*), applications are first translated into **C** to be used as input for the model checking tool. Their approach is tailored towards an MSP430 processor. The description of the system hardware is performed at the level of pins and registers of the processor, which are modeled using non-deterministic statements. External devices such as sensors are not considered for verification.

The main contribution of [BK11b] is the application of partial order reduction (POR) (see Section 2.2.5) at the level of **C** statements to reduce the increased state space size, introduced by interrupt modeling. When interrupts based systems are verified all potential locations for an interrupt occurrence have to be considered. Therefore, the interleaving model for parallel executions is used (cf. Section 2.2.5). As an interrupt can occur at any time and can suspend the original application, calls to the ISR are added at each statement of the original application. In contrast to normal parallel program executions for which the interleaving model is normally used, an ISR itself cannot be interrupted by the main application. The approach assumes that an interrupt can occur after each **C** statement, whereby atomic statements are defined

as colon-terminated **C** statements. Depending on the processor architecture this can correspond to several assembler assignments. Therefore, the approach abstracts the actual behavior of interrupts from the hardware of a specific processor to the level of **C** statements. Furthermore, no assumptions are made how often interrupts can occur, e.g. whether they occur periodically.

To reduce the number of calls to the interrupt service routine introduced by the interleaving model, Bucur and Kwiatkowska apply POR. POR has been used before in model checking tools, which support the verification of multi-threaded software such as the SPIN model checker. To perform POR, **C** statements within the application are assumed as transitions, which change the states of the program, whereby the call to an ISR is seen as a parallel statement, which can be executed at any time. The general idea behind POR is described in Section 2.2.5. Bucur and Kwiatkowska implemented their version of POR for **C** applications for the model checking tool CBMC, also used in this work. This is possible for all properties of CBMC as it only supports safety properties of the kind $\mathbf{G}f$.

In practice, POR corresponds to reducing interrupt calls to locations, where the ISR actually affects the behavior of the execution of the applications. These are locations, where variables shared between the ISR and the application are read or written. An example of POR for **C** applications is given in Section 4.4.2.

A similar approach to deal with interrupt modeling is presented by Schlich et al. in [SNBB11]. They propose a technique called *Interrupt Handler Execution Reduction*, to reduce the number of interrupt call interleavings, which is implemented for their *[MC]SQUARE* model checker. Their approach is also inspired by POR, however, the implementation is done at the level of assembler code, which is the input of *[MC]SQUARE*. This model checker supports several microcontroller architectures for which models of the hardware at the level of registers exist. Properties which shall be checked are specified using the temporal logic **CTL**, whereby the next operator **X** is not allowed to enable the reduction of interrupts[14].

An enhancement of the POR modeling of Bucur and Kwiatkowska is given in [KLM+15] for systems, which use multiple interrupts with priorities. It is assumed that interrupts occur between **C** statements as in [BK11b]. Based on *partial-order encoding*, dependencies between interrupts based on their priorities are checked and impossible executions are discarded. In contrast to Bucur and Kwiatkowska not only software for an MSP430 microcontroller and *TinyOS* is regarded, however, no hardware modeling is discussed. To perform evaluation with multiple interrupts a wrapper function is used, as discussed in the next section.

[14] This restricts properties to such, which describe stuttering equivalent paths (Definition 2.11), similar to the definition of $\mathbf{LTL}_{\neg\mathbf{X}}$ (Definition 2.12).

```
1  unsigned int current_time;
2  unsigned int min_delay_time, max_delay_time;
3  ...//Initialization
4  while (1) {
5    current_time = clock_time();
6    statement_1;
7    statement_2;
8    ...
9    statement_n;
10   assert( ((current_time + min_delay_time) <= clock_time()) &&
11           (clock_time() < (current_time + max_delay_time)) );
12 }
```

Figure 4.7: Example application that can be interrupted.

4.4.2 Applying Existing Approaches to the Verification of *Contiki* Applications

To discuss the existing approaches for modeling interrupts and how they are suited for the verification of *Contiki* applications a general example, which is shown in Figure 4.7, is used. The application shall demonstrate how applications depend on interrupts and how they affect the program flow and therefore, which assertions can be verified. This example uses the timer interrupt shown in Figure 4.6, which is called by the hardware periodically after a certain number of microcontroller clock cycles have passed. As already described, the interrupt uses the variables `count` and `seconds` to store the current system time. A short description of the application follows:

At the beginning of the while loop (lines 4-12), the current system time is stored in the variable `current_time` (line 5) using the *Contiki* pre-defined clock API function `clock_time`. The function returns the current system time (value of `count`). Afterwards, n arbitrary program statements are executed (lines 6-9), which do not access the variables used to store the system time in the interrupt. The assertion to be checked on the system is given in lines 10-11 and checks whether the overall system time has increased in the interval defined by the variables `min_delay_time` and `max_delay_time` (with `min_delay_time` < `max_delay_time`), compared to the value stored in `current_time`. This means that the assertion only can be proven valid, when during the execution of the n arbitrary statements the number of interrupts defined by the delay time interval occur. When too few or too many interrupts occur, verification will fail.

One way of modeling of interrupts for formal verification (see Section 4.4.1), based on the interleaving model, is to add a call to the interrupt statement at every point of code, where the interrupt could occur in the real system. The transformed example

```
 1 unsigned int current_time;
 2 unsigned int min_delay_time, max_delay_time;
 3 ...//Initialization
 4 while (1) {
 5   if (!nondet_0()) // non-deterministic call
 6     timer_interrupt();
 7   current_time = clock_time();
 8   if (!nondet_0())
 9     timer_interrupt();
10   statement_1;
11   if (!nondet_0())
12     timer_interrupt();
13   statement_2;
14   ...
15   if (!nondet_0())
16     timer_interrupt();
17   statement_n;
18   if (!nondet_0())
19     timer_interrupt();
20   assert( ((current_time + min_delay_time) <= clock_time()) &&
21           (clock_time() < (current_time + max_delay_time)) );
22 }
```

Figure 4.8: Example application instrumented with interrupt calls after each statement.

```
1  unsigned int current_time;
2  unsigned int min_delay_time, max_delay_time;
3  ...//Initialization
4  while (1) {
5    if (!nondet_0()) // non−deterministic call
6      interrupt_wrapper();
7    current_time = clock_time();
8    statement_1;
9    statement_2;
10   ...
11   statement_n;
12   if (!nondet_0())
13     interrupt_wrapper();
14   assert( ((current_time + min_delay_time) <= clock_time()) &&
15           (clock_time() < (current_time + max_delay_time)) );
16 }
```

Figure 4.9: Example application after reduction of interrupt calls with POR.

application of Figure 4.7 is shown in Figure 4.8. Between each statement of the original program, a call to the interrupt function `timer_interrupt` is added. This call is made non-deterministic using the function call `nondet_0`[15], modeling that an interrupt can, but does not need to occur.

It can be seen that the calls to the interrupt routine increase the program size significantly. POR can now be used to reduce the size of the program. After applying POR the interrupt handler functions are only called at points where an interrupt actually effects the flow of the program, as described in Section 4.4.1. For the example application, the results of this reduction are shown in Figure 4.9. The non-deterministic calls to the interrupt handler function can be reduced to the statement, where the system time is stored, before the calculation starts (lines 5-6) and when the assertion on the system runtime is checked (lines 12-13). Only in these cases the variable `count` - which is changed by the interrupt - is accessed by the `clock_time()` function.

When applying POR, the information how often the ISR can be called from the application is lost (the possible number of ISR calls becomes independent of the actually executed statements). In the case of the example application after reduction, the ISR could be only called up to 2 times, whereas in the original application the number of possible interrupt calls depends on the number of statements n. Therefore, an interrupt wrapper function has to be used as shown in Figure 4.10. This wrapper function allows it to call the actual interrupt function arbitrarily often, at every point of code where the wrapper is being called. As noted in [KLM+15] the overall number of possi-

[15] `nondet_0` is available as part of CBMC, other model checking tools offer similar constructs to model non-deterministic variables.

```
1 void interrupt_wrapper(void) {
2   while (1) {
3     if (!nondet_0())
4       timer_interrupt();
5     else
6       return;
7   }
8 }
```

Figure 4.10: Interrupt wrapper function used for POR.

ble interrupt occurrences in the system can be controlled by restricting the `while(1)` loop in line 2 of Figure 4.10.

The described modeling approach for interrupts is based on the assumption that an ISR can occur after each program statement. Looking at the assertion that shall be proven in Figure 4.7 this assumption is not sufficient. The actual runtime of the program on the system is not taken into account for determining the number of possible interrupt occurrences. However, the assertion that is verified checks that the interrupt handler is called during the calculation only a certain number of times.

When non-deterministically calling the interrupt handler at all possible program points as shown in Figure 4.8, the property can only be proven if the specified interval for the delay is between 0 to $n+1$, as this is the number of possible interrupt calls. When applying POR as done in Figure 4.9 with the interrupt wrapper shown in Figure 4.10 verification will always fail, as the `count` variable can be increased arbitrarily often in the interrupt function.

This leads to the following observations for the modeling of interrupts within the *Contiki* system:

- The use of POR together with an infinite number of interrupts is a safe over-abstraction of the interrupt behavior as zero to infinite interrupts are checked. When an application passes verification for such a modeling style it can be assumed safe. However, the modeling can introduce spurious counterexamples, as the number of assumed interrupts and time of occurrence are not possible on the real system. Furthermore, when using BMC it is not possible to check the correctness of an infinite `while(1)` loop (see also Section 5.1.3).
- The interrupt modeling with POR doesn't take into account any information, how often an interrupt handler could be called from the program. This is related to principle of stuttering described in Section 2.2.5. As noted in [CGP00]:

> "Stuttering is a particular interesting concept for asynchronous systems because there is no correlation between the time separating two events and the number of transitions occurring between them."

- When using POR, it is not possible to statically set a bound how many interrupts can occur as the number of program statements that appear between calls to an interrupt function cannot be determined statically.

The consequence of these observations is that it is not possible to verify applications with POR, whose correct behavior depends on the frequency of interrupt occurrences. This includes programs which rely on the appearance of interrupts, as interrupts might never occur when using POR[16]. Furthermore, properties that directly or indirectly check how often an interrupt can occur cannot be verified. This, e.g. corresponds in *Contiki* to checking how much time has passed in the system using the event timer system. Another assertion which cannot be proven on the system is, e.g. that the size of the *Contiki* event queue, described in Section 3.3, is sufficient, as events can be created based on the number of interrupt calls. To verify programs whose correct behavior depends on the frequency of interrupt occurrences a new modeling approach, *periodic interrupt modeling* (PIM), is suggested.

4.5 *Periodic Interrupt Modeling* - Taking System Runtime into Account

```
1  static unsigned int statement_counter;
2  unsigned int interrupt_period = 10;
3
4  void initialize_interrupt(void) {
5    statement_counter = nondet_int();
6    __CPROVER_assume(statement_counter < interrupt_period);
7  }
8
9  void periodic_interrupt(void) {
10   statement_counter++;
11   if (statement_counter == interrupt_period) {
12     timer_interrupt();
13     statement_counter = 0;
14   }
15 }
```

Figure 4.11: Functions used for *periodic interrupt modeling*.

[16] An example of this problem in practice is shown in the example in Section 6.4.

Periodic interrupt modeling (PIM) allows it to verify assertions that rely directly or indirectly on the number and frequency of interrupt invocations, as the information how often an interrupt occurs on the actual system is considered during modeling.

The implementation of the interrupt modeling style uses two functions `initialize_interrupt` and an interrupt wrapper `periodic_interrupt`. These functions are shown in Figure 4.11. In addition, it uses the global variables `statement_counter` and `interrupt_period`, which are of type unsigned (always positive) integers. The global variable `statement_counter` is used to count the number of statements since calling the interrupt handler the last time. The variable `interrupt_period` is used to define the period between the actual interrupt handler calls. This period is given as the number of **C** statements between the calls to the interrupt handler, as expected in the real system[17].

The function `initialize_interrupt` shall be called as the first statement of the program to be verified (inside the **C** `main` function). Its purpose is to initialize the `statement_counter` to an arbitrary value $0 \leq$ `statement_counter` $<$ `interrupt_period`. This simulates the behavior that at system startup it is not known how many cycles it will take till the first interrupt appears. By using a non-deterministic variable it is made sure that the state space is searched for all possible values in the interval.

The checking whether the interrupt handler needs to be called is implemented in the interrupt wrapper function `periodic_interrupt`. This function is called after each **C** statement of the original program and therefore at the beginning the variable `statement_counter` is incremented (line 10 in Figure 4.11). When the statement counter reaches the value of `interrupt_period`, the actual interrupt handler shown in Figure 4.6(b) is called (line 12) and the counter is set back to zero (line 13).

The transformed example application (cf. Figure 4.7) for interrupts is shown in Figure 4.12. At the beginning of the program, the function `initialize_interrupt` is called (line 3). Afterwards, a call to the interrupt handler for PIM `periodic_interrupt` has been added between each **C** statement of the application. Depending on the chosen interrupt distance `interrupt_period` and the actual executed statements n between storing `current_time` (line 7) and checking the assertion in line 16, verification will either pass or fail.

Discussion The main practical challenge for interrupt modeling is choosing the correct modeling style. POR has the advantage that it is easily applicable, without additional knowledge about the modeled interrupts. It is a safe over-abstraction of the actual interrupt behavior, as all possible interrupt occurrences are considered.

[17]Using a variable instead of a constant allows it to modify the period of the interrupts during runtime.

```
1  unsigned int current_time;
2  unsigned int min_delay_time, max_delay_time;
3  initialize_interrupt();
4  ...//Initialization
5  while (1) {
6    periodic_interrupt();
7    current_time = clock_time();
8    periodic_interrupt();
9    statement_1;
10   periodic_interrupt();
11   statement_2;
12   ...
13   periodic_interrupt();
14   statement_n;
15   periodic_interrupt();
16   assert( ((current_time + min_delay_time) <= clock_time()) &&
17            (clock_time() < (current_time + max_delay_time)) );
18 }
```

Figure 4.12: Example application transformed using PIM.

This means when an application passes verification with POR it can be assumed safe. However, POR leads to executions, which are not existing on the real system and can lead to assertions which cannot be verified, as discussed in Section 4.4.2.

Verification using PIM is more accurate as the interval of interrupt occurrences is considered and can be applied to periodically occurring interrupts. Therefore it is possible to verify applications, which rely on a specific number of interrupts. Assertions can be proven, which check that a certain number of interrupts occurred. One practical challenge when applying PIM is calculating the period in which interrupts occur and translating them to the number of **C** statements. The occurrence of periodic interrupts is usually defined by a sampling time e.g. for sensors or the period of a watchdog timer. Based on the clock frequency of the processor running the application, it can be easily calculated after how many processor clock cycles a periodic interrupt occurs. Mapping these clock cycles onto **C** statements is however not easily possible, as each statement can be mapped onto several assembler commands. Each assembler statement can then take a varying number of clock cycles. The specified interrupt period can therefore always only be an estimate.

Although PIM has been introduced in the context of *Contiki*, the method can be applied to all kinds of embedded software relying on periodic interrupts. However, a typical scenario that highlights the importance of correct interrupt modeling is the event-driven kernel of *Contiki*, as the modeling of interrupts has a direct influence on the order in which processes are executed.

Furthermore, it has to be noted, that in this thesis only systems with one active interrupt are considered. In principle, both PIM and POR can be extended to be used together with multiple interrupts. For PIM the interrupt wrapper `periodic_interrupt` shown in Figure 4.11 can be extended to handle multiple interrupts, by introducing different interrupt periods for each periodic interrupt. The POR based approach can handle multiple interrupts independently by adding calls to program locations, where the interrupts affect the program behavior. As shown in [KLM+15], further challenges such as interrupt priorities have to be considered, when dealing with multiple interrupts. Furthermore, verification times increase significantly when considering multiple interrupts.

5

Chapter 5

Model Checking and Verification Flow

This chapter describes, which steps are necessary to verify a *Contiki* application using the model checking tool CBMC. The general idea of the model checking approach is to treat the **C**-based hardware description of the system, the operating system source code, and the source code of the applications as input to the model checker. Section 5.1 explains, which parameters are used for model checking and how loop unwinding is performed for the *Contiki* system. The in this thesis developed verification framework for *Contiki* applications is presented in Section 5.2. This framework can then be used to compare the POR and PIM interrupt modeling approach (see Chapter 6). Finally, in Section 5.3 it is discussed how the verification framework can be applied for test case generation.

In the overall model checking process, described in Section 2.2.1, this corresponds to the *running phase*, as it is described how the model checking tool CBMC is used and which parameters are being set.

5.1 Bounded Model Checking and Setting Loop Bounds

When applying BMC a bound for unwinding loops has to be set, as described in Section 2.2.4. This bound determines how often a loop can be executed and therefore limits the search within the state space. It has a direct influence on the size of the resulting SAT formula.

When using the CBMC tool for trivial examples no bound must be set as CBMC can determine - using static analysis - the number of needed loops unwindings. However,

this is not possible in general, especially for cases where the bound depends on the value of a non-deterministic input variable. Furthermore, in many applications and especially embedded systems, unbounded loops exist as periodic tasks are performed, which do not terminate. Determining a sufficient bound, which includes all possible system behaviors, corresponds to finding a *completeness threshold* for bounded model checking. This is a hard problem as already discussed in 2.2.3.

Other approaches to extend BMC to prove completeness with induction based techniques are shown amongst others in [SSS00], [GLD10] and [DHKR11]. However, as induction requires a certain depth for the loop unwinding of an infinite loop, which shall be handled, these approaches suffer also from the state space explosion problem and are probably not suited for the verification of realistic *Contiki* applications. A further examination regarding their performance and applicability is out of scope for this work.

5.1.1 Unbounded Loops in *Contiki*

In the *Contiki* operating system, the main loop of the kernel is unbounded, as the system shall run endlessly. The execution of *Contiki* with all applications running on the system can be seen as a sequential program, whose length can by restricted by setting the number of unwindings of the main scheduler loop, which controls the number of events processed in the system. In *Contiki*, only one application is running at a time, which is possible due to the use of cooperative multitasking, and the *protothread* programming model, where applications are written in a cooperative way using `PROCESS` macros.

In *Contiki*, all loops, except the main scheduler loop in the kernel, must terminate after some time, as they would otherwise block the kernel from starting other processes. Therefore, *Contiki* application processes must cooperatively return control to the kernel, so that other processes can run[18]. As already discussed in Section 3.4 and shown in Figure 3.4 it is possible to suspend applications using *local continuations*, whereby the return point of the application is stored and later used to resume the application.

As an example, consider the *Contiki* application in Figure 5.1. In this application, an event timer is set and it waits for a time of one second till the timer expires using the `PROCESS_WAIT_EVENT_UNTIL` macro. The original code written by the user is shown in Figure 5.1(a) and the code after expansion of the *protothread* macros in Figure 5.1(b). The `while(1)` loop in lines 8-11, Figure 5.1(a), always terminates after 1 iteration

[18]By inspecting the number of loop unwindings for an application as shown in Section 5.1.2 it can be detected that an application does not return control to the kernel.

```
1  #include "contiki.h"
2  PROCESS(loop_process, "loop");
3  AUTOSTART_PROCESSES(&loop_process);
4  PROCESS_THREAD(loop_process, ev, data)
5  {
6    PROCESS_BEGIN();
7      static struct etimer et;
8      while(1) {
9        etimer_set(&et, CLOCK_SECOND);
10       PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
11     }
12   PROCESS_END();
13 }
```

(a) Original application with a seemingly unbounded loop.

```
1  #include "contiki.h"
2  PROCESS(loop_process, "loop");
3  AUTOSTART_PROCESSES(&loop_process);
4  PROCESS_THREAD(loop_process, ev, data)
5  {
6    PROCESS_BEGIN();
7      static struct etimer et;
8      while(1) {
9        etimer_set(&et, CLOCK_SECOND);
10       do {
11         PT_YIELD_FLAG = 0;
12         (process_pt)−>lc = 10; case 10:;
13         if((PT_YIELD_FLAG == 0) || !(etimer_expired(&et))) {
14           return 1;
15         }
16       } while(0)
17     }
18   PROCESS_END();
19 }
```

(b) Application with `PROCESS_WAIT_EVENT_UNTIL` macro expanded.

Figure 5.1: *Contiki* LED API and implementations for hardware access and verification.

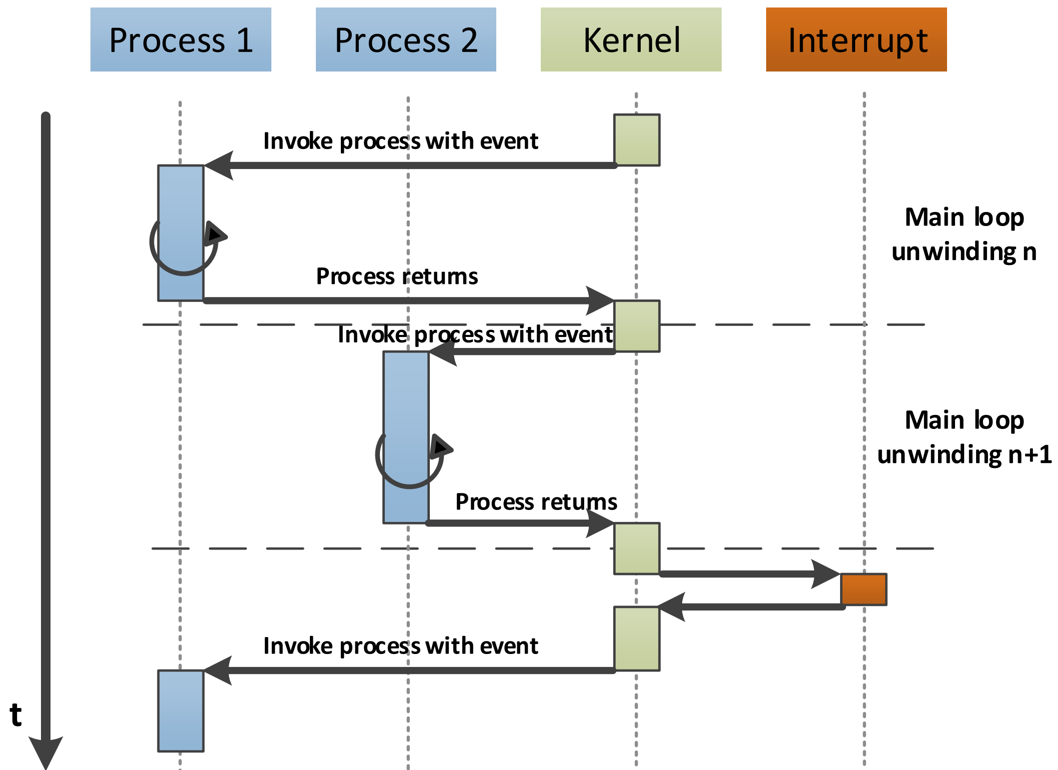


Figure 5.2: Unwinding of loops and restriction of the overall main loop unwinding.

due to the return statement in line 14 of Figure 5.1(b). The program is resumed at the point of the `case`-statement in line 12 when the application is invoked using an event, by using the local continuation stored in this line. The label *10* used in the `case`-statement is derived from the original line number of the *protothread* macro in Figure 5.1(a). When the condition is not fulfilled the application returns (line 14). Due to the addition of `return`-statements in the *protothread* macros, the seemingly unbounded loops within *Contiki* processes are actually bound.

Under the assumption, that all *Contiki* processes are bounded due to the use of *protothreads* (i.e. they will always give control back to the kernel), it is possible to verify the applications running on a *Contiki* system by bounding only the main loop of the *Contiki* scheduler. This corresponds to restricting the `while`-loop in line 6 in Algorithm 3.1 shown in Section 3.3. By bounding the unwindings of the main loop it is possible to set the unwinding depth at a global point, without modification of the applications, which are verified. In Figure 5.2 the general loop unwinding principle is depicted. With each additional unwinding, another event can be executed from the event queue and therefore the search depth for the verification is increased.

5.1.2 Setting Bounds

To verify an application with BMC and to determine the bounds for unwinding loops (*unwinding depth* cf. Section 2.2.4) Algorithm 5.1 is applied. This is possible as the CBMC tool allows it to set individual loop unwindings for each loop in the system. For the description of the algorithm the following definitions are used:

Definition 5.1. A ***loop unwinding*** is a tuple $U = \langle u_{name}, u_{depth} \rangle$ where

- u_{name} gives the name of the loop. Each loop in the program has a unique name.
- $u_{depth} \in \mathbb{N}$ determines the unwinding depth, i.e. the maximum number of times the loop can be executed.

Definition 5.2. A *list* of all loop unwinding tuples U belonging to a *Contiki* system is denoted by $\mathcal{U}$.

```
1  𝒰 ←getLoops() ; /* Get a set of all loops in the system and assign it to the list of
   loop unwinding tuples */
2  foreach U ∈ 𝒰 do
3      U.u_depth ← 1 ;                                   /* Set initial unwinding limit */
4  while (true) do
5      BMCResult ← RunBMC(𝒰) ;      /* Run model checking tool with current
       unwinding */
6      if BMCResult = Unwinding assertion failed then
7          foreach U ∈ 𝒰 do
8              if U.u_name = Name of failed unwinding assertion then
9                  U.u_depth ← U.u_depth + 1 ;        /* Increment unwinding bound for
                   failed assertion */
10                 if  U.u_depth > Loop unwinding limit then
11                     return Verification failed
12     else if BMCResult = Program assertion fails then
13         return BMCResult ;                           /* Return counterexample */
14     else if BMCResult = Verification successful then
15         return Verification successful
```

Algorithm 5.1: Pseudocode algorithm for running CBMC and unwinding loops.

In line 1 of Algorithm 5.1 the list $\mathcal{U}$ is declared and initialized with the list of all loops in the program, which is requested from the model checking tool and determined using a static code analysis. Afterwards an initial loop unwinding bound is set in line 2 for

all loops. After this initialization the actual verification is performed in the `while` loop from line 4 - 15 until verification fails or succeeds. Thereby, the model checking tool is started repeatedly (line 5) using the current loop unwindings U. Three cases are possible when running the model checker:

- **Case 1:** Verification ends with a failing unwinding assertion (line 6). In this case, the unwinding depth for this loop is incremented (lines 7 - 9) and the model checker is started again. However, when a global loop unwinding limit is reached for unwinding, verification is canceled (lines 10-11).
- **Case 2:** Verification ends with an assertion failing, which is not an unwinding assertion (line 12). In this case, a counterexample is returned, which leads to a violation of a property. As loops are incremented only by one, the counterexample is also the shortest possible counterexample.
- **Case 3:** Verification succeeds with no assertion failing (line 14).

As the main loop of the *Contiki* kernel is restricted, the algorithm terminates with either a failing assertion (case 2) or with no violated assertion (case 3), as long as there is no process, which blocks the overall system, i.e. it does not return control to the *Contiki* kernel (cf. Section 5.1.1). In case 1 the loop unwinding is increased till a global loop unwinding limit is reached. This allows it to detect *Contiki* processes that contain loops that require loop unwindings higher than the global loop unwinding limit and are potentially blocking the system. When no such limit is set unwinding would continue, in the case of an unlimited loop within a process, until the model checking tool would run out of memory, i.e. the verification would not terminate. If verification fails, because the unwinding limit is reached, then the loop that reaches the limit has to be inspected, to check whether the unwinding limit is not sufficient or the loop actually blocks the system.

From a practical point of view, an initialization of the loop tuples with an initial value of 1 as shown in line 2 is often not needed. The initial number of loop unwindings for bounded loops can be taken often from the application source code by manual inspection. Furthermore, the result of the loop unwinding from one verification run can be reused as a starting point for re-running the verification after fixing a bug in the program.

However, the shown algorithm allows it to run the verification without any user interaction, except for setting the main loop bound and will determine the shortest possible counterexample. Running the program with too many loop unwindings has no effect on the verification result, but it increases the overall verification runtime.

```
1  void interrupt_wrapper(void)
2  {
3    int por_count = 0;
4    while (por_count < por_limit) {
5      por_count++
6      if (!nondet_0()){
7        timer_interrupt();
8      }
9      else
10       return;
11   }
12 }
```

Figure 5.3: Restricting the number of possible interrupt calls for BMC when applying POR.

5.1.3 Loop Unwinding and Interrupt Modeling

For the modeling of interrupts as discussed in Section 4.4 also loop unwinding has to be considered. Similar to any other parts of the code, interrupts service routines can contain loops. In contrast to normal code an ISR must exit as they would block the complete system, leaving it in an unusable state. Typically, the code of ISRs have a very short runtime, as they perform service tasks and should not suspend normal system operations for a long time. When modeling interrupts, the calls to an ISR are turned into a normal function call. Therefore, the loop unwinding algorithm as described in Section 5.1.2 can be used and loops within interrupts will be unwound normally.

A special case is the use of the POR interrupt modeling style. In this modeling style, a safe over-abstraction of the system behavior is achieved by allowing an arbitrary number of interrupt calls to occur (see Figure 4.10). Such a `while(1)` loop would, however, make loop unwinding impossible: Unwinding assertions would always fail, because always an execution path exists, in which the `return` statement cannot be reached. Therefore, the `while` loop, must be restricted for BMC. In Figure 5.3 the variable `por_limit` restricts the number of possible calls to the ISR which is modeled.

For POR however, it is still possible, that an interrupt might never occur as the call to the actual ISR is guarded by a non-deterministic variable (lines 6-8 in Figure 5.3). This leads to problems when the termination of a loop depends on the occurrence of an interrupt. Such non-terminating loops can be detected by limiting the number of possible loop unwindings, as presented in Algorithm 5.1. An example application where this problem occurs is described in Section 6.4.

When using PIM no special measures for loop unwinding are necessary, as no additional

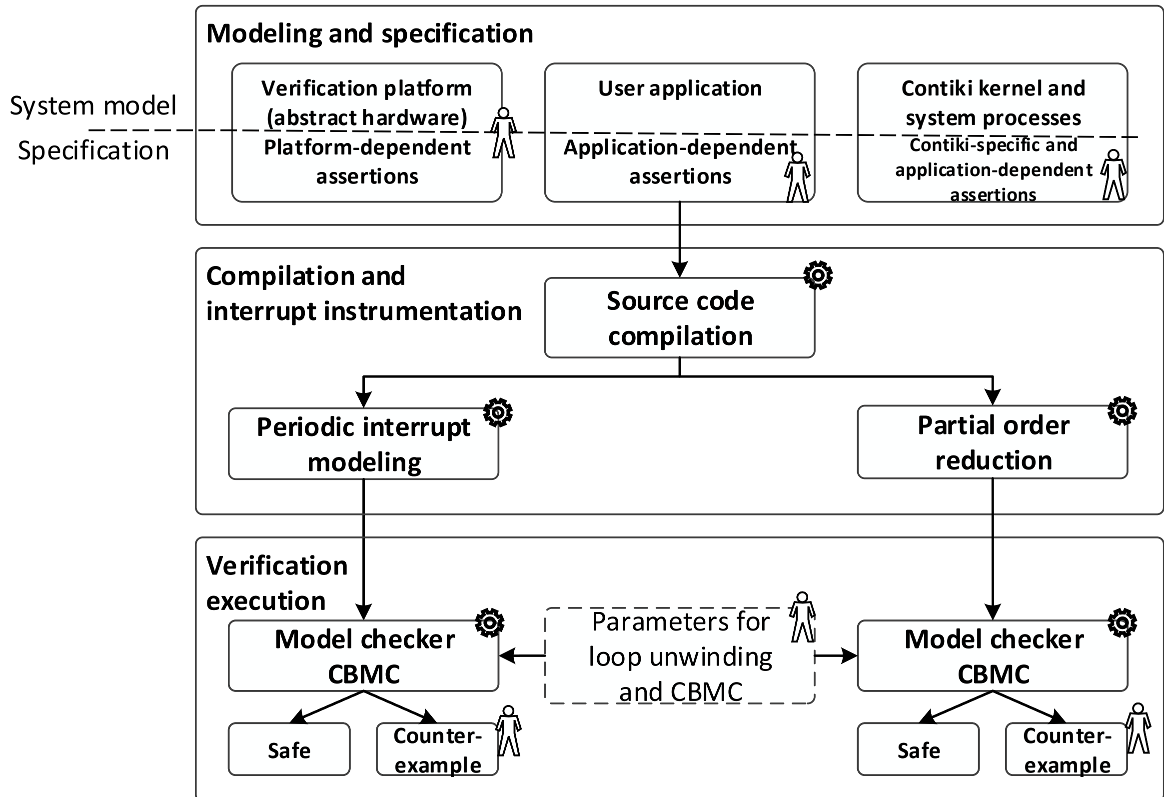


Figure 5.4: Verification framework for Contiki applications.

loops are introduced in the interrupt wrapper functions (see Figure 4.11).

5.2 Verification Flow and Implemented Tools

Figure 5.4 shows an overview of the verification flow for *Contiki* applications, as implemented in this work. The verification flow can be divided basically into three phases, a *Modeling and specification* phase, a *Compilation and interrupt instrumentation* phase, and a *Verification execution* phase. A description of the necessary steps and tools for each phase and the level of automation is given in the following sections.

5.2.1 Modeling and Specification Phase

This phase requires the most manual effort that is spent to prepare the overall system for verification. Starting from the *specification* and the use cases of the embedded system assertions have to be derived, which shall be checked formally (cf. Section 4.2).

Furthermore, the *system model* as input for the model checker must be built.

The system modeling and specification phase can therefore be split into three parts:

- Creation of an abstract hardware description of the verification platform and the corresponding assertions. The hardware of the embedded system is modeled by replacing drivers of hardware components with models suited for verification written in **C**, which can include *platform-dependent assertions*. Additionally, also models for system interrupt service routines must be written as shown in Section 4.4. In this way a hardware platform is built, which can be reused for several applications, making this process only necessary when new hardware components are added to an embedded system (cf. Section 4.3 and Section 4.4). The created models for such a platform can then be added to *Contiki* platforms described in Section 3.5.
- Preparation of the user application: In this step, the *Contiki* applications written using *protothreads* are adapted for verification. *Application-dependent assertions* may be added, which are then verified together with automatically generated assertions (cf. Section 4.2).
- Preparation of the *Contiki* kernel and system processes. In this step, the system is configured so that only used drivers for an application are loaded[19]. Furthermore, *Contiki-specific assertions* can be added, which check the correct usage of *Contiki* APIs. Depending on the application, additional checks e.g. for the size of the event queue might be activated. Similar to the creation of the hardware description the modeling process often needs only to be performed once for each hardware platform (cf. Section 4.2).

From the verification point of view, the hardest part is to find the correct assertions to be checked and to model the hardware at the level of abstraction needed to verify the applications. When a verification platform was built, it can then be used to verify different applications running on the same hardware.

From the application point of view the general restrictions for software model checking described in Section 2.2.4 must be considered, especially regarding the supported **C** standard libraries supported by CBMC. In all examples of Section 6 no modifications were required. However, the complexity of the overall systems can make it necessary to modify an application e.g. by reducing the number of loop iterations or the sizes of buffers.

[19] When performing verification, a possible recursive call within the *Contiki* event timer was found, which can not be restricted using the loop unwinding switches available by CBMC. This call was manually unwound and guarded by an assertion, which, however, was not violated in any experiment performed in Chapter 6. No other changes in the *Contiki* kernel and libraries have been made.

5.2.2 Compilation and Interrupt Instrumentation

After the source code has been prepared, it has to be compiled and instrumented as shown in Figure 5.4. Therefore, CBMC provides a special compiler called *goto-cc*, which can be used as a replacement for the GNU C Compiler (GCC) compiler, that is often used to build *Contiki* executables. Therefore, in a first step using *goto-cc* all sources needed for the verification of a *Contiki* system (verification platform sources, application sources, and *Contiki* kernel system sources) are compiled into a single executable for verification (a so-called *goto-binary*). This is comparable to the normal flow for the creation of *Contiki* sensor node binaries, which are after compiling transferred to the sensor node hardware.

The created *goto-binary* can be used as unaltered input for verification with CBMC. However, when the system uses interrupts a second step for interrupt instrumentation is necessary depending on the modeling style:

- POR based modeling: In this case, the implementation provided by CBMC, which implements the POR algorithm of Bucur and Kwiatkowska, discussed in Section 4.4.1, is used. For the transformation of the compiled *goto-binary*, CBMC provides the *goto-instrument* tool. The ISR function has to be provided as an argument, whereby calls to this function are added automatically in the code at places where the execution of the program is influenced. It must be noted, that the by CBMC provided implementation does not support calling functions from within an ISR. Therefore, the functions of the *Contiki* kernel, which are called within ISRs must be inlined manually. Otherwise, a function call from the ISR could lead to a recursive call of the ISR, as the function could again call an interrupt.
- PIM based modeling: In this case, the pre-processed code of a *goto-binary* is dumped as **C** code and used as input for a *Python*-based program, which performs the transformation for PIM as described in Section 4.5. As the implementation works on the pre-processed code of the *goto-cc* compiler, the implementation of the transformation is simplified. The code only uses a subset of **C** statements where e.g. switch statements have been removed. The resulting code of the transformation is then translated back into a *goto-binary* and is used for verification. Similar to the interrupt instrumentation with POR, function calls within an ISR must be inlined, so that no recursive ISR calls occur.

All compilation steps have been implemented using *make*-files and scripts and can be performed completely automatically so that no user interaction is necessary. The parameters, which must be provided for transforming the program are the names of the ISRs, as well as the number of interrupts in the case of POR and the interrupt

period in case of PIM. The so build *goto-binary* is used as input for verification in the next step.

5.2.3 Verification Execution

During the verification execution phase shown in Figure 5.4 the *goto-binaries* generated in the compilation and interrupt instrumentation phase (cf. Section 5.2.2) are used as input for CBMC. The model checker is started repeatedly to perform the actual verification using the BMC algorithm. When running CBMC, verification parameters must be set, such as the loop unwinding, the kind of system architecture[20], as well as backend options like the used SAT solver. Furthermore, the type of the assertions, which are generated automatically by CBMC can be chosen (cf. Section 2.2.4 and Section 4.2).

The algorithm to perform loop unwinding described in Section 5.1.2 has been implemented in a *Python* program, which allows it to start CBMC automatically. Verification stops when an assertion violation within the program is found or the main loop is unwound to the specified limit and no bugs were found. In the case of a failing assertion, a counterexample is generated. It has to be examined, to find out and understand, whether the error is a problem introduced due to inaccurate modeling or a real problem within the system. After a modeling problem has been fixed, verification can be repeated. Depending on the location of the problem the already used looped unwinding set can be reused (cf. Algorithm 5.1).

The third possible outcome of verification is the hardest to tackle: Verification takes "too long" or CBMC runs "out of memory"[21]. In this case, the application size must be reduced e.g. by restricting the sizes of arrays or by simplifying the application.

5.3 Test Case Generation Using Bounded Model Checking

SAT-based test case generation has been a standard technique for generating test patterns to detect faults in digital circuits for a long time [SBSV96] and is comparable to the use of BMC for safety properties. BMC is able to generate minimal counterexamples to properties, when the search space bound k gets incremented one by one

[20] CBMC supports, amongst others, different kinds of bit-widths for integers, modes for floating point calculations and endianness settings.

[21] In all experiments performed in this work (see Chapter 6) no "out of memory"-error occurred, as verification was stopped before, due to long runtimes.

starting from an initial state (see Section 2.2.3). Such a counterexample can be also seen as a test case, as it contains an execution trace, which brings the system into a certain state. Therefore, BMC can be used to generate test cases by specifying the system state, which shall be covered using an assertion.

Test cases for software can be useful for different purposes, e.g. for reaching coverage criteria for certain safety certifications [RUPP12]. In the context of *Contiki* applications, test case generation can be useful for debugging and understanding a system, as questions like: "How might a certain register achieve it's value?" or "Which input values lead to turning on the LED?" can be answered. This is especially useful in combination with virtual prototypes of the system hardware (see Section 2.1.1). On a virtual prototype, the generated test case trace can be further simulated to better understand the system behavior.

To perform test case generation a certain code section, which shall be reached by the test case must be marked in the code. When using CBMC, this can be done by adding an `assert(0)` statement at the location. The model checker will then try to find a counterexample to violate the assertion. If this assertion can be violated, the counterexample states all inputs, which have to be applied to the system to reach the marked code segment.

Performing the steps described in Section 5.2, test case generation can be performed similarly to proofing that an assertion must not be violated. In contrast to verification, the goal, however, is to generate a counterexample. As BMC is used, the generated counterexample trace is minimal regarding the number of loop iterations, when the loop unwinding begins from 0 for all loops.

Similarly to verification also test case generation has to consider how interrupts occur within a system and the corresponding interrupt modeling style.

- POR is not suited for test case generation when applied for a system, which used periodically occurring interrupts e.g. for modeling timers. The notion that an interrupt can occur at any time (cf. Section 4.4.2), and with no specified interval leads to test cases, which cannot be reproduced on real hardware or in a simulation.
- In contrast PIM accurately models the occurrences of interrupts (cf. Section 4.5). The generated test cases are realistic regarding the timing behavior and therefore the behavior of the real system hardware.

An example of test case generation for *Contiki* applications is given in Section 6.6.

Chapter 6
6 Verification of *Contiki* Applications

This chapter demonstrates how *Contiki* applications can be verified based on the verification approach described in Chapter 4 and Chapter 5. At this, special attention is paid on comparing the different interrupt modeling techniques. From the model checking process, this chapter corresponds to the *running phase* and especially the *analysis phase*, as the results of running the model checker are analyzed and presented. All shown examples are based on real-world applications, which were run on a TI-MSP430 based microcontroller, as supported by *Contiki* with additional drivers added for LCD and acceleration sensors. In total five example applications were examined, which are either part of the *Contiki* open source release or applications developed for the embedded system platform:

- *Hello World* example (part of *Contiki*): The "Hello World" application demonstrates the general way of verifying *Contiki* applications.
- *LED Blink* example (part of *Contiki*): This is the example application described in Section 3.2, which uses the event timer system together with interrupts.
- *LED Fader* example (part of *Contiki*): This is a more complex application, where the LED of an embedded system fades using interrupts. This example also uses busy waiting.
- *Bubble sort* with LCD example: The application demonstrates how an algorithm can be verified by model checking using non-deterministic inputs and custom assertion. In addition, the use of platform-specific assertions and user-defined assertions is shown.
- *3-axis acceleration* sensor: Here, the *Contiki* sensor system works together with interrupts to periodically collect data from a sensor. It is also shown how model

checking can be used for test-case generation.

As already noted the examples were originally executed on the *Contiki* target for the MSP430 platform. To verify the examples using CBMC, a verification platform was built as described in in Chapter 4 and shown in Figure 4.1. Therefore, the drivers needed to run the applications in the original hardware platform were replaced with abstract models for verification. The actual implementation of the verification platform is based on the *Contiki native* hardware platform, which is included in the *Contiki* release and allows it to run *Contiki* applications under Linux to test hardware independent applications. Based on this platform the driver models, which are used within the applications were added including a model of the timer interrupt.

This chapter is structured as follows. In Section 6.1 the setup for the verification and the used parameters are explained. In Section 6.2-6.6 the verification results for each application are shown. A summary and discussion of the verification results is given in Section 6.7.

6.1 Experimental Setup

All verification runs were performed using CBMC tool version 4.9 on *Red Hat Enterprise Linux 7* running on a *DELL PowerEdge R630* server with 2 *Intel Xeon E5-2637v3 processors* (4 cores with 3.5 GHz and 15MB cache) and 512 Gb RAM. The shown verification times always refer to a run with all internal loops of an application already unwound so that no unwinding assertions occur (see Section 5.1). Only the execution of the main *Contiki* scheduler loop is limited to the given value. The number of times this loop is unwound, therefore, limits the search space of the system and the depth within which bugs can be found.

When using the *partial order reduction* based interrupt modeling style the maximum number of possible interrupt occurrences is stated. For the experiments, the number was set exemplarily to POR=2 and POR=10 to show the influence on the runtime. Similarly, when using *periodic interrupt modeling* the distance between interrupt occurrences is stated in the number of **C**-statements as described in Section 4.5. For the experiments, values of PIM=2 and PIM=200 were chosen exemplarily. To automatically generate the PIM and POR version of the application and to run CBMC, the verification flow described in Section 5.2 was used.

For automatically generated assertions the pointer check and division by zero checks were activated (see Section 2.2.4). In addition, unwinding assertions were always activated making sure that all loops were unwound sufficiently. Furthermore, the bitwidth for integer variables was set to 16-bit. *Lines of code* for applications refer to the

```
1 PROCESS(hello_world_process, "Hello world process");
2 AUTOSTART_PROCESSES(&hello_world_process);
3
4 PROCESS_THREAD(hello_world_process, ev, data)
5 {
6   PROCESS_BEGIN();
7   printf("Hello, world\n");
8   PROCESS_END();
9 }
```

Figure 6.1: The *Hello World* application.

application before transformation with POR or PIM, as reported by the CBMC tool.

The shown verification times are split into unwinding/preprocessing time as well as SAT solving time. SAT solving time is reported by the CBMC tool directly. The sum of these two times is the overall verification time needed for the specific CBMC invocation.

The *size of the program expression* is the size of the program after unwinding of loops given in steps of the static single assignment form (see Section 2.2.4), before simplification and generation of the SAT formula are performed. The size of the SAT formula is given by clauses and variables when using the MiniSAT solver backend of CBMC.

6.2 *Hello World* Application

The first example to be examined is a simple *Hello World* application, running on the verification platform. The source code of this application is shown in Figure 6.1. The application does not use any special hardware drivers and neither the event timer system. Furthermore, it only consists of one *Contiki* process, which runs after the invocation to completion. The application can therefore be seen as a simple test, which checks the correctness of the *Contiki* kernel, to make sure that no programming errors are present (at least for a system executing no other processes). Although no interrupts are used by this application, the verification platform is still configured to use interrupts. Therefore, interrupts are modeled using the PIM and the POR based approaches. The process function of the transformed *Hello World* application for POR and PIM is respectively shown in Appendix A.1.1 and A.1.2. It can be seen that for POR no calls to an ISR are necessary, as the program doesn't interact with any program variables used by interrupts. In the case of PIM, the statement counter has to be increased after each call, therefore making it necessary to call the corresponding

interrupt wrapper. It has to be noted, that only the transformed function of the application process is shown. To verify the program also the *Contiki* kernel is used together with the driver models for the verification platform.

Verification results The verification results are shown in Table 6.1 for checking automatically generated assertions. The complete verified application including the *Contiki* kernel has a size 424 of lines of code, 680 automatically generated assertions were checked. No errors were found in the program, which is expected as the application is very simple and the *Contiki* source code mature. Overall verification times are short and it takes only some seconds to verify the program.

When comparing the verification times and SAT problem size for PIM it can be seen, that the size of the SAT problem is basically identical for different chosen interrupt periods of 2 and 200 statements. This behavior is expected as the program size after unwinding is the same and the only difference in the programs are the check on the number of statements when an interrupt occurs (see Section 4.5).

When using POR and comparing the results for a maximum of 2 and 10 possible interrupt occurrences an increase in runtime number of program steps (and SAT problem size) can be seen. The reason is that the state space increases significantly with the number of possible interrupt executions, as the loop, which limits the possible number of interrupt execution has to further be unwound (see Figure 5.3).

A special case happens when looking at verification times with increasing main loop unwindings. In Table 6.1 it can be seen that with increasing the number of possible main loop unwindings (1 vs. 5), verification time and SAT problem size is constant for both interrupt modeling styles and corresponding parameters. When analyzing in detail the output messages generated by the CBMC tool during a preprocessing process for the program an increasing number of statements is removed from the program by static analysis leading to an only slight increase in the size of the program expression steps. This is possible, as the *Hello World* program is executed only once during the first main loop unwinding, where it runs to completion and the *Contiki* process ends. For further main loop unwindings, no process is running in the system as the application has finished.

6.3 *LED Blink* Application

This example shows verification results for the *LED Blink* example introduced and extensively discussed in Section 3.2. In contrast to the *Hello World* example in Section 6.2, this application uses the event timer system and the *Contiki* LED API. It is

lines of code	424							
number of assertions	680							
main loop unwindings	1				5			
interrupt modeling style	POR	POR	PIM	PIM	POR	POR	PIM	PIM
max. number of timer interrupts	2	10	-	-	2	10	-	-
timer interrupt period	-	-	2	200	-	-	2	200
program expression steps	4,568	11,688	11,362	11,362	4,616	11,736	11,950	11,950
variables	87,125	313,907	119,357	119,355	87,125	313,907	119,357	119,355
clauses	201,661	974,124	328,519	328,513	201,661	974,124	328,519	328,513
unwinding and preprocessing in seconds	0.572	3.300	1.037	0.988	0.613	3.143	1.109	1.183
SAT solving in seconds	0.342	2.170	0.220	0.205	0.369	2.107	0.206	0.217

Table 6.1: Verification times and problem sizes for *Hello World* example verifying automatically generated assertions.

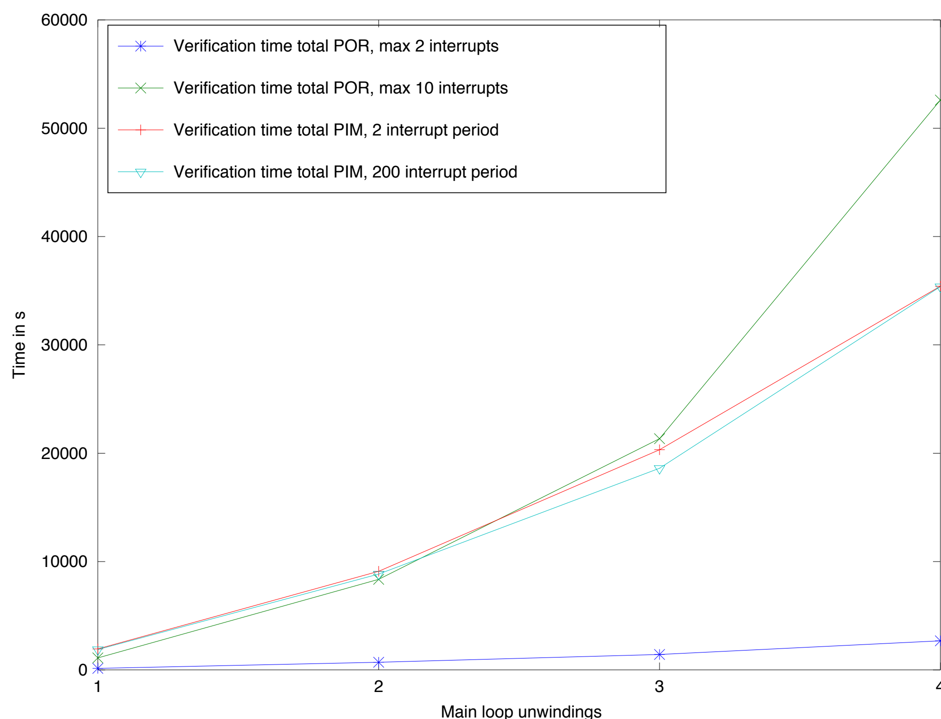


Figure 6.2: Verification times for increasing main loop unwinding for the *LED Blink* example, verifying automatically generated assertions for POR and PIM.

a typical *Contiki* application, which performs a task periodically.

Verification results Detailed verification results for the *LED Blink* example are given in Table 6.2. Although the size of the program and the number of assertions to be checked is quite similar to the *Hello World* example the verification results and times differ. Verification times are mostly in the range of several minutes and can go up to several hours for larger main loop unwindings.

In Figure 6.2 the overall verification times with increasing main loop unwindings are shown for POR and PIM modeling style. In contrast to the *Hello World* example, when increasing the main loop unwindings the overall verification time increases with each unwinding of the main loop. With each main loop unwinding the *LED Blink* application is called from the *Contiki* kernel, and a potentially different behavior might occur. When doubling the main loop unwinding it can be seen that the program expression steps also nearly double. For this example, CBMC is not able to detect automatically, whether further loop unwindings are needed to check the correctness of the program.

Similar to the verification results for the *Hello World* example the interval of interrupts, when using a PIM based modeling has no effect on the verification time and

lines of code	433							
number of assertions	692							
main loop unwinding	2				4			
interrupt modeling style	POR	POR	PIM	PIM	POR	POR	PIM	PIM
max. number of timer interrupts	2	10	-	-	2	10	-	-
timer interrupt period	-	-	2	200	-	-	2	200
program expression steps	116,324	336,005	990,272	990,272	228,544	656,969	1,940,446	1,940,446
variables	9,421,675	16,731,045	12,652,230	12,652,228	22,714,669	37,396,527	26,951,148	26,951,146
clauses	38,422,398	64,195,791	44,988,981	44,988,975	94,916,688	147,216,759	98,433,795	98,433,789
unwinding and preprocessing in seconds	503.82	5,340.14	8,823.07	8,570.22	1,900.96	19,011.10	34,501.58	33,831.13
SAT solving in seconds	201.95	3,020.30	280.19	277.98	780.37	33,586.80	915.96	940.97

Table 6.2: Verification times and problem sizes for *LED Blink* example verifying automatically generated assertions.

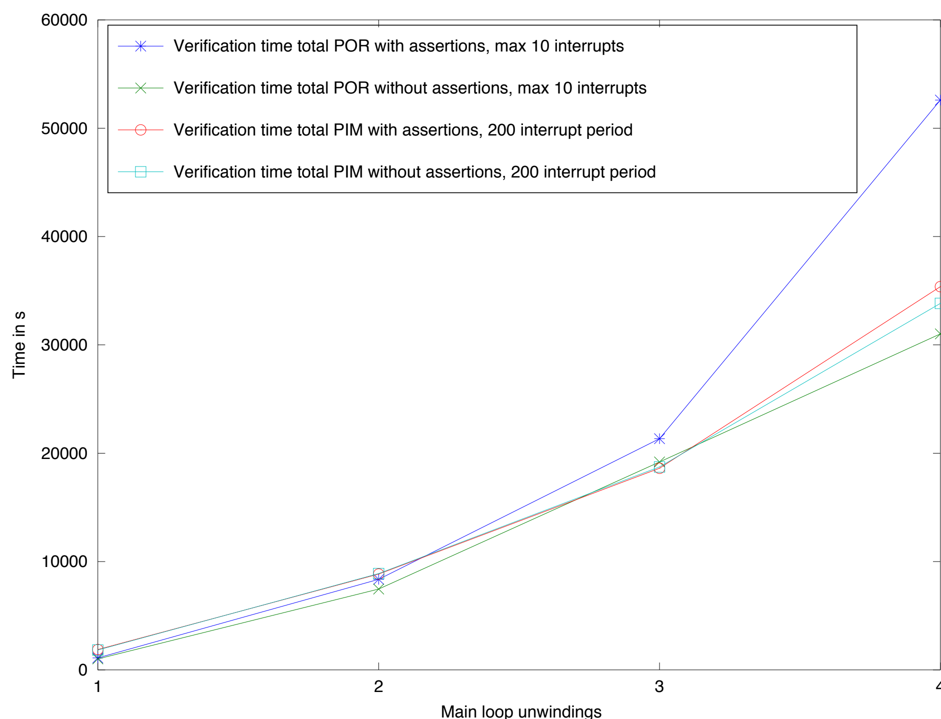


Figure 6.3: Verification times for increasing main loop unwinding for the *LED Blink* example, comparing automatically generated assertions with no assertions for POR and PIM.

size of generated SAT problem. For the POR interrupt modeling style the number of possible interrupts increases the verification time significantly, leading to longer verification times than when using the PIM approach. This example shows that the number of possible interrupts leads for POR to a significant increase in verification times, and possibly making it slower than the PIM approach.

The influence on the verification time of adding automatically generated assertions, compared to no assertions (only loop unwinding) is shown in Figure 6.3. For POR, a significant increase of verification time for automatically generated assertions can be seen, especially in the case of a larger number of main loop unwindings. For PIM the influence of automatically generated assertions is only small.

In Figures 6.4-6.7 it is shown how much time is spent by CBMC in the SAT solving phase and for unwinding the program and preprocessing the formula. For POR with a maximum of 2 interrupts it can be seen that most of the verification time is spent in the loop unwinding phase. However, for 10 interrupts and increasing main loop unwindings SAT solving takes more and more time. With 3 main loop unwindings about half the verification time is spent in unwinding and half in SAT solving. With the main loop unwinding of 4 about two third of the verification time is spent for SAT

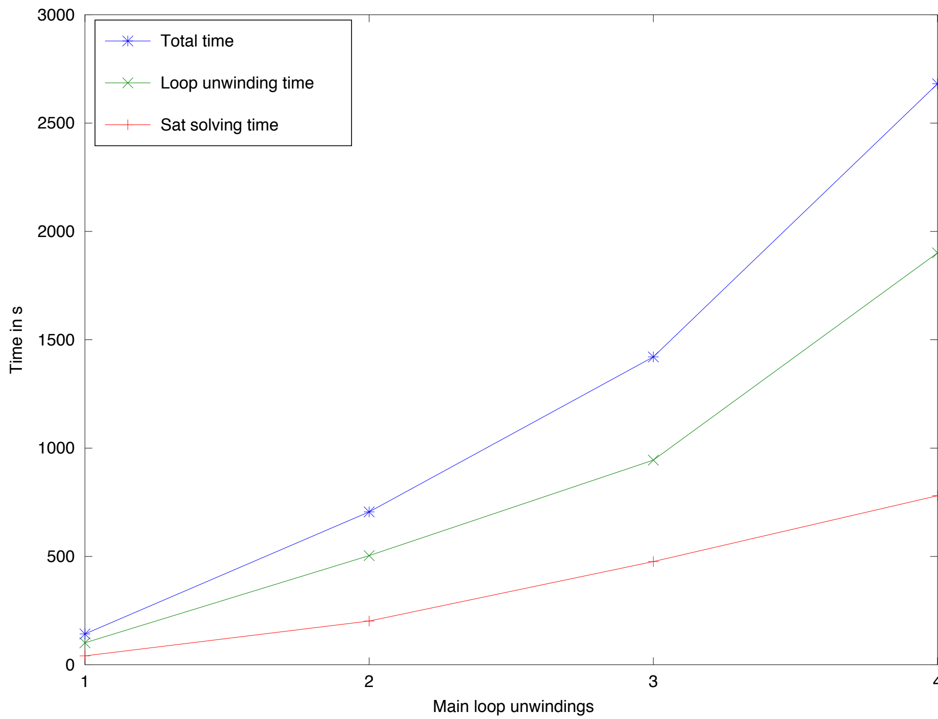


Figure 6.4: Verification times split into SAT solving and loop unwinding for increasing main loop unwinding for the *LED Blink* example, with POR = 2.

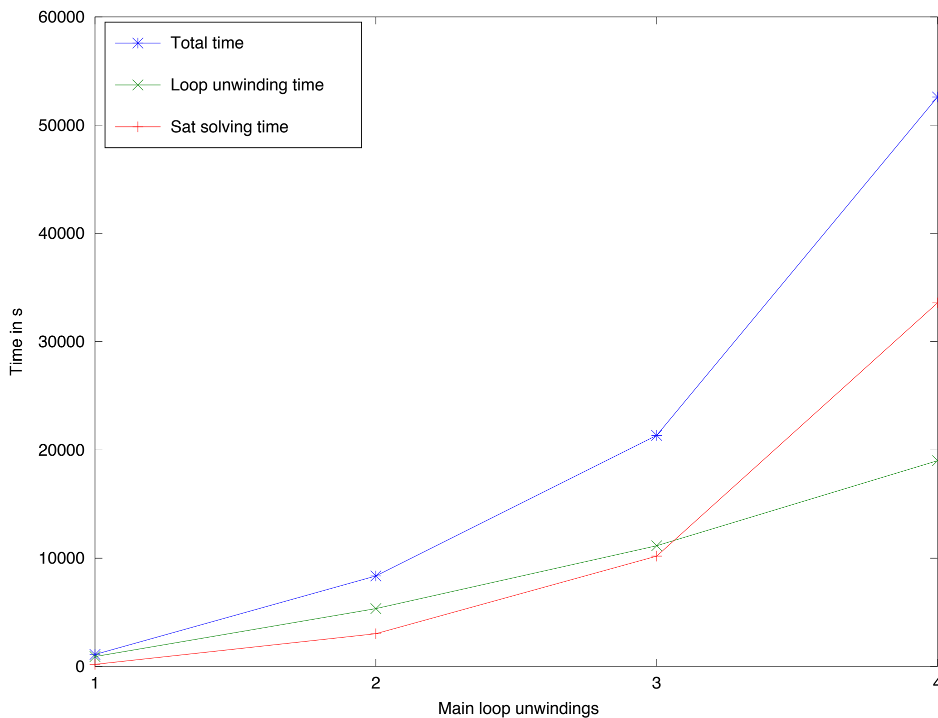


Figure 6.5: Verification times split into SAT solving and loop unwinding for increasing main loop unwinding for the *LED Blink* example, with POR = 10.

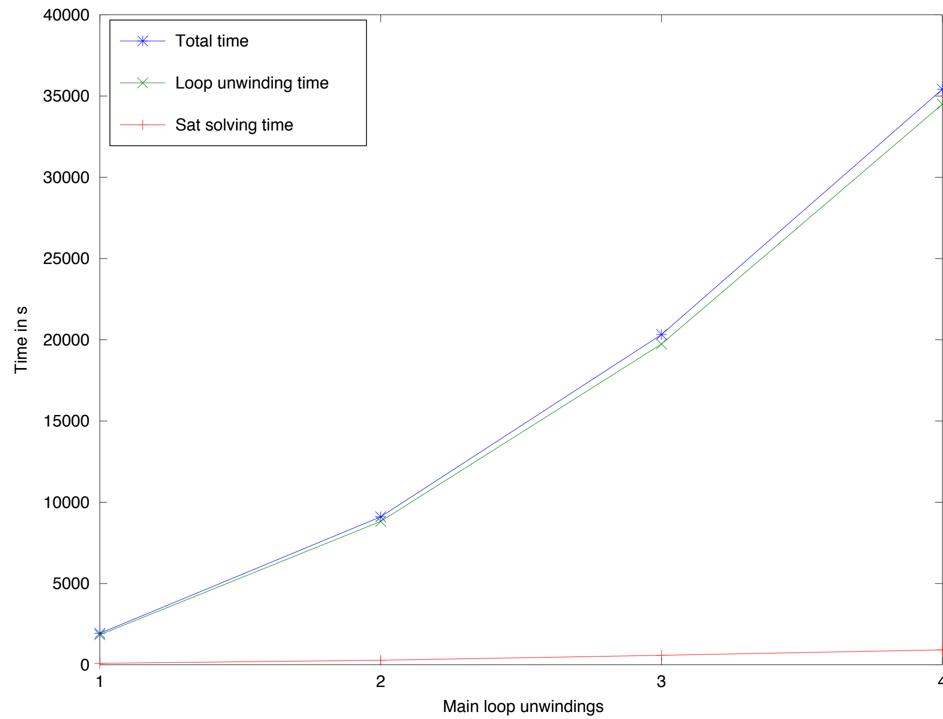


Figure 6.6: Verification times split into SAT solving and loop unwinding for increasing main loop unwinding for the *LED Blink* example, with PIM = 2.

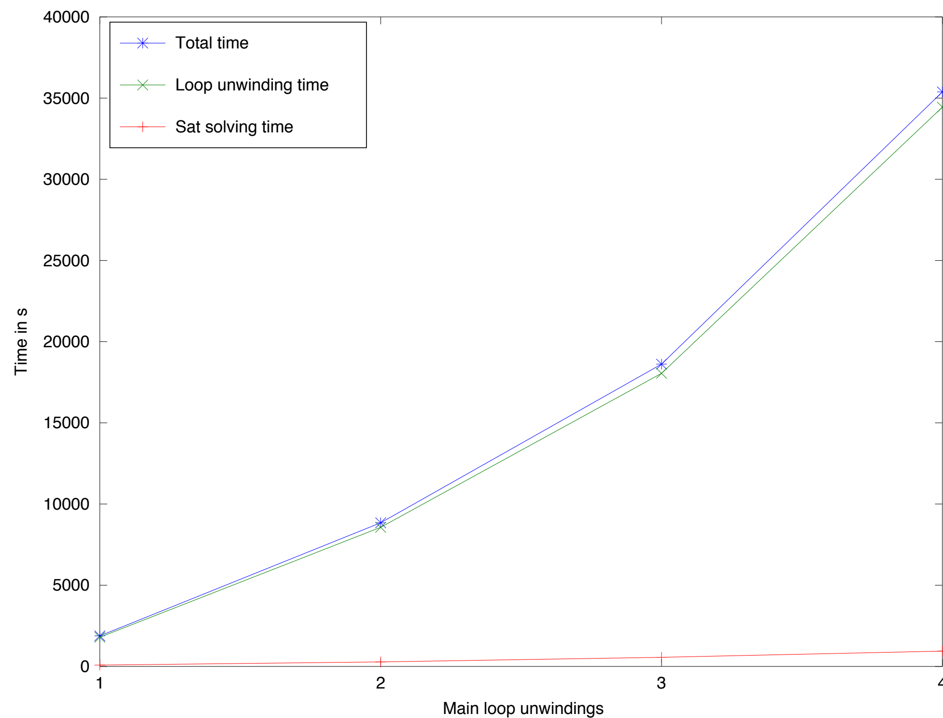


Figure 6.7: Verification times split into SAT solving and loop unwinding for increasing main loop unwinding for the *LED Blink* example, with PIM = 200.

solving. The formula is becoming so large that efficient solving is not anymore possible and SAT solving gets the bottleneck in verification.

For PIM, however, most of the time is spent in preprocessing and only a small amount of time in SAT solving. This result is interesting, as overall verification times between PIM and POR seem similar. For POR most time is spent in SAT solving as when using PIM the main time is spent in loop unwinding. The reason is, that for PIM the initially unrolled program is much larger as there are more statements in the program after interrupt instrumentation compared to POR (a possible interrupt call is added after each statement). Processing this program into a SAT formula therefore takes more time than in the case of POR. This process is however very efficient, although the program expression in the case of PIM is about 3 times larger than for POR with 10 interrupts, the resulting formula is actually smaller and therefore SAT solving faster than for POR.

The comparison shows that the size of the unwound program is not a measure to judge the actual runtime. Even the much larger program generated for PIM can be efficiently reduced. It shows when a larger number of interrupt occurrences has to be considered that the PIM approach can be faster than the POR approach.

Overflow of the event queue So far only automatically generated assertions were verified. To demonstrate that the verification of a timing related property is not possible using POR (assertions which rely on the number of interrupts called as described in Section 4.4) the *LED Blink* application was slightly modified (see Section A.1.3). Instead of creating 1 event timer in the original *LED Blink* application 4 event timers are used, which trigger the blinking of the LED. The property which shall be proven is that the size of the Contiki event queue is sufficient, meaning that when $|\mathcal{E}^{MaxEvents}| = MaxEvents$ no new event is generated. Therefore an *event queue full* assertion was added to the *Contiki* kernel, which verifies that no event is posted when the queue is full. When the queue is full posting new events is not anymore possible, having severe consequences on the overall system. The implementation of the assertion is shown in Figure 4.2(b) and discussed in Section 4.2.

The system was configured to a $MaxEvents$ of 1, i.e. only one event timer can be waiting in the system to be processed. When checking this property with POR it was configured, so that at least 4 interrupts can occur, so that all event timers can be triggered. However, in this case, the event queue can be full, as it is possible that all interrupts trigger at the same time, which is not possible on the real system (see Section 4.5). Therefore, when trying to verify this application with POR verification fails, as the *event queue full* assertion can be violated with a counterexample. The cause is that POR over-abstracts the interrupt behavior, so that interrupts can occur

main loop unwinding	2	
interrupt modeling style	PIM	POR
max. number of timer interrupts	-	10
timer interrupt period	200	-
program expression steps	1,540,627	394,515
variables	28,898,788	15,598,988
clauses	108,029,589	61,145,896
unwinding and preprocessing in seconds	23,825	8,418 [1]
SAT solving in seconds	1,121	4,423 [1]

[1] Time for generating a counterexample

Table 6.3: Verification time and problem size for the *LED Blink* example with *event queue full assertion*.

consecutively. The generated counterexample, however, can never occur on a real system leading to a *false positive* - a counterexample, which is not possible on the real hardware and is caused by imprecise modeling. These counterexamples are hard to recognize and lead to an additional debugging effort.

In contrast, when using PIM with an interrupt period of 200 statements, verification passes, as enough time exists for events of the event timer system to be processed before another timer event occurs. However, when choosing a too small period of interrupts in the case of PIM it is also possible, that the *event queue full assertion* fails. This example demonstrates the usefulness of the PIM approach, and that it can be applied to real-world systems.

In Table 6.3 the results are summarized. These results show that for this application it is possible in reasonable time to find a counterexample in the POR case, whereas for PIM no counterexample exists, as the property does not get violated. Furthermore, this application demonstrates, that it is very hard to predict the SAT solving performance. In the PIM case the formula for SAT solving is larger with regard to variables and clauses, still solving is faster than for POR, where a counterexample exists.

6.4 *LED Fader* Application

The *LED Fader* application is also part of the base *Contiki* examples and periodically fades-in and fades-out an LED of an embedded system. It demonstrates the use of the *protothreads* programming model to spawn and resume processes. The source code of the application is shown in Appendix A.1.4. In this application *protothreads* with the `PT_*` macros are used (cf. Section 3.4 and Appendix A.2) and both the timer system (where no event is created when a timer expires), as well as the event timer system are used.

Furthermore, in this application, interrupts play an important role, as busy waiting is performed to wait for an event. Therefore, when not modeling interrupts correctly the application may not terminate.

The application uses two *protothreads* called `fade_in` (lines 19-35) and `fade_out` (lines 37-53). The brightness of the LED is adjusted by switching a LED on and off, where the time the LED is switched on either increases (`fade_in`) or decreases (`fade_out`). A `for` loop is used to call the *Contiki* function `clock_delay`, which delays the system a certain number of clock cycles. The two *protothreads* `fade_in` and `fade_out` are controlled by a *protothread* `fade` (lines 55-70), which first calls the `fade_in` *protothread* and afterwards the `fade_out` *protothread*. The *protothread* call using `PT_SPAWN` is *blocking* and only returns when the spawned *protothread* exits (in contrast to `PT_SCHEDULE`, which calls a *protothread* and returns when the spawned process yields). Therefore, first, the `fade_in` *protothread* increases the brightness of the LED and afterwards, `fade_out` decreases the brightness of the LED. Afterwards, the `fade` thread is suspended for a time period using the event timer system (lines 65-66) and afterwards returns to call `fade_in` and `fade_out` using a while loop.

The main process of the application (`fader_process`) is shown in lines 81-116. To allow reuse of the *protothreads* `fade_in`, `fade_out` for different LED colors, the *protothread* `fade` is parametrized using the control structure of type `fader`. This control structure (lines 10-15) stores information for the three fade *protothreads*. An instance of the structure is created as a static variable (as no stack is saved for *protothreads*) to fade an LED (a red LED is used for this application). Afterwards, the *protothread* is initialized (line 89). Now the LED is faded in and out by scheduling the `fade` *protothread*. When the *protothread* returns within 1 second it is started again to continue fading. As no event is created when using the function `timer_set`, busy waiting is performed to wait for the timer to expire (lines 93-96).

The main loop of the fader example is shown in lines 101 to 116. Here fading is repeated when the static variable `onoroff` is enabled (lines 110-113). By calling the `process_poll` function the application process is automatically invoked again when the fader is enabled. Furthermore, the process is scheduled to run periodically using the event timer system (lines 105-108).

As described, the fader process can be switched on and off by modifying the static variable `onoroff`. This variable can be modified using the functions `fader_on` and `fader_off`.

Verification results Without modifications, this application can only be verified using *periodic interrupt modeling*, due to the busy waiting used in lines 93-96. This loop only ends when a certain number of interrupt calls have occurred, leading the

lines of code	487
number of assertions	837
main loop unwinding	2
interrupt modeling style	PIM
timer interrupt period	2
program expression steps	3,490,119
variables	41,304,656
clauses	149,227,006
unwinding and preprocessing in seconds	99,296
SAT solving in seconds	5,369

Table 6.4: Verification time and problem size for the *LED Fader* example verifying automatically generated assertions using PIM.

timer to expire, making it mandatory to model interrupts. Without interrupts, the application never terminates and blocks the system.

When using POR the application cannot be verified, as there is always an execution possible, where this loop never ends, as all possible calls to the interrupt function are guarded with `!nondet_0()` and may never be executed, as shown in Figure 4.9 and discussed in Section 4.4.2. Therefore, in the case of POR loop unwinding for this loop never terminates i.e. a loop unwinding assertion for this loop is always violated.

In contrast, when using PIM the interrupt function gets always called after a certain number of statements, therefore making it possible to verify the application. In Table 6.4 verification results for the application are shown, with the main loop unwinding of 2 and PIM enabled. As the size of the application is quite big it shows the limits of model checking for software, with over a day of verification time. When comparing verification times with the largest verified PIM program of the *LED Blink* application (see Table 6.2, main loop unwinding = 4), it shows that number of variables and clauses of the generated SAT formula is about 50 % larger. This increase in SAT formula size leads to about three times of the overall verification time (104665 seconds compared to 34772 seconds). As expected no violations were found for the checked automatically generated assertions.

6.5 *Bubble sort* with LCD Application

To verify the correct usage of an LC-Display driver and to demonstrate how *application-dependent assertions* can be added, a *Contiki* application was written, which sorts an array of numbers in ascending order using the bubble sort algorithm and checks using an assertion that the sorting was performed correctly. The sorted numbers are afterwards written out on an LC-Display. The size of the verification problem can

for this application be increased with the size of the array, which shall be sorted. In Appendix A.1.5 the source code of the application is shown. To add the assertions a **C** pre-processor macro `VERIFY` was added which is only set, when the application is compiled for formal verification[22]. In addition to the automatically generated assertions also *application-dependent assertions* and *platform-dependent assertions* are checked. The *application-dependent assertions* check the correctness of the sorting algorithm and the *platform-dependent assertions* check whether the LC-Display is correctly driven by the application. These assertions include checks, that the LCD is correctly initialized, only valid symbols are displayed, and no symbols are displayed at an index position outside of the display. A description of the application follows.

The application uses the following functions which are called from the main *Contiki* application process (lines 78-120):

- The function `printInteger` (lines 6-25) prints an integer number on the LC-display. As the driver supports only displaying single digits the number has to be decomposed, before it is displayed using the function `lcd_disp_char` (line 17). Assertions are used to make sure that the decomposition is performed correctly.
- The function `getIntArray` (lines 26-40) creates an array of values to be sorted, whereby the parameter `nmax` determines the size of the array. In the case of the variant used for verification non-deterministic values are used (line 35), which makes sure that the verification is performed for all possible integer values.
- The function `bubbleSort` (lines 43-63) sorts the passed array using the bubble sort algorithm, whereby the parameter `n` determines the size of the array.
- The function `checkResultArray` (lines 65-76) makes sure that the resulting array is sorted correctly by looping over the passed array.

The main process thread (lines 81-120), which uses the described functions then just initializes the LC-Display (lines 93-95), gets the numbers to be sorted (line 98), prints the unsorted numbers (lines 100-103), sorts the numbers, checks whether sorting was performed correctly (lines 109-112), and finally prints the sorted numbers (lines 115-118).

Verification results The *Bubble Sort* example application is used to demonstrate, how the size of the array, which is sorted influences the overall verification time. Therefore, the array to be sorted using the *Bubble Sort* application was increased using the macro `BUBBLE_SORT_NMAX` and compared for both POR and PIM. As the

[22]When the macro is not set, the code will not be compiled. This makes it possible to reuse the same code for compilation for the target platform and for verification.

lines of code	473			
number of assertions	680			
problem size n	5		10	
interrupt modeling style	POR	PIM	POR	PIM
max. number of timer interrupts	2	-	2	-
timer interrupt period	-	2	-	2
program expression steps	13,998	111,840	30,888	261,180
variables	397,063	1,247,110	770,627	2,885,124
clauses	1,254,395	3,901,360	2,566,812	9,232,972
unwinding and preprocessing in seconds	3.94	80.09	22.20	499.78
SAT solving in seconds	5.47	13.42	52.33	79.34

Table 6.5: Verification time and problem size for the *Bubble Sort* example verifying automatically generated assertions and *application-* and *platform-dependent assertions.*

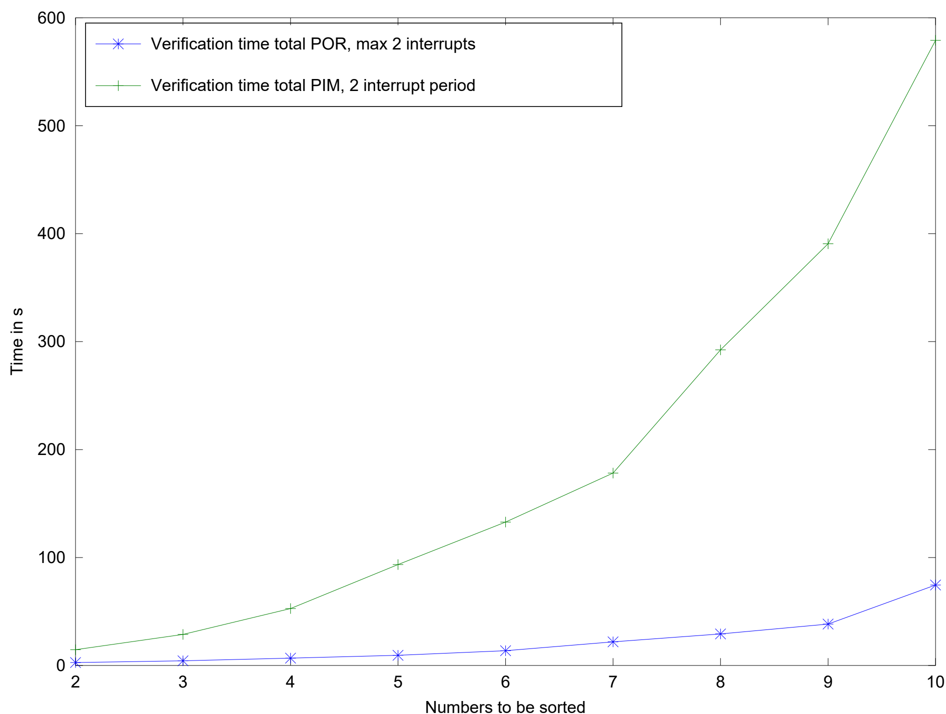


Figure 6.8: Verification times with increasing array sizes for the *Bubble Sort* example, verifying assertions for POR and PIM.

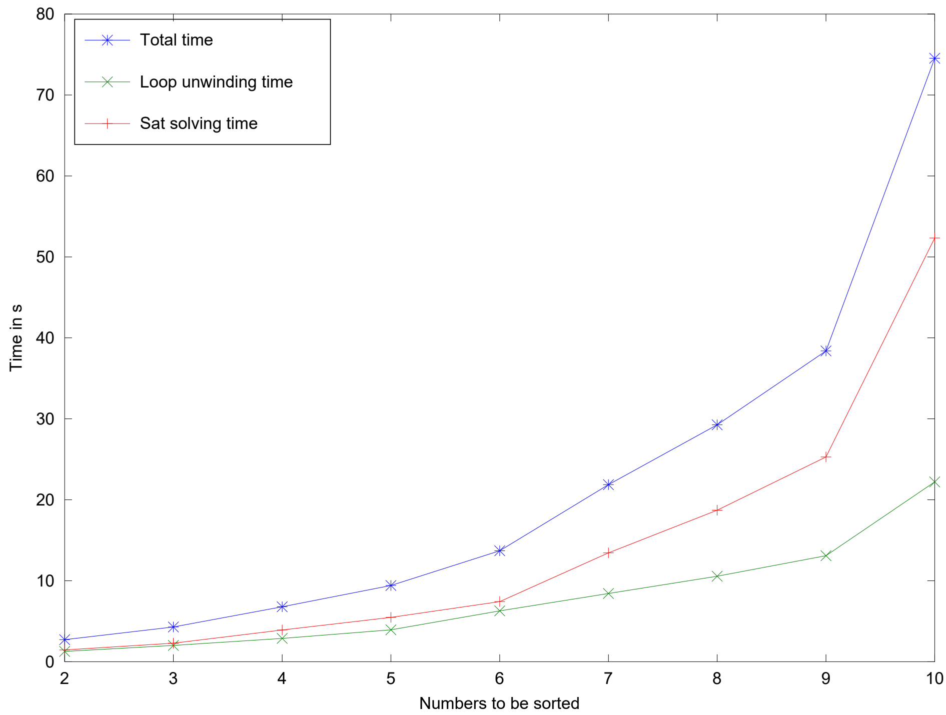


Figure 6.9: Verification times split into SAT solving and loop unwinding, increasing the array of to be sorted numbers n for *Bubble Sort*, with POR = 2.

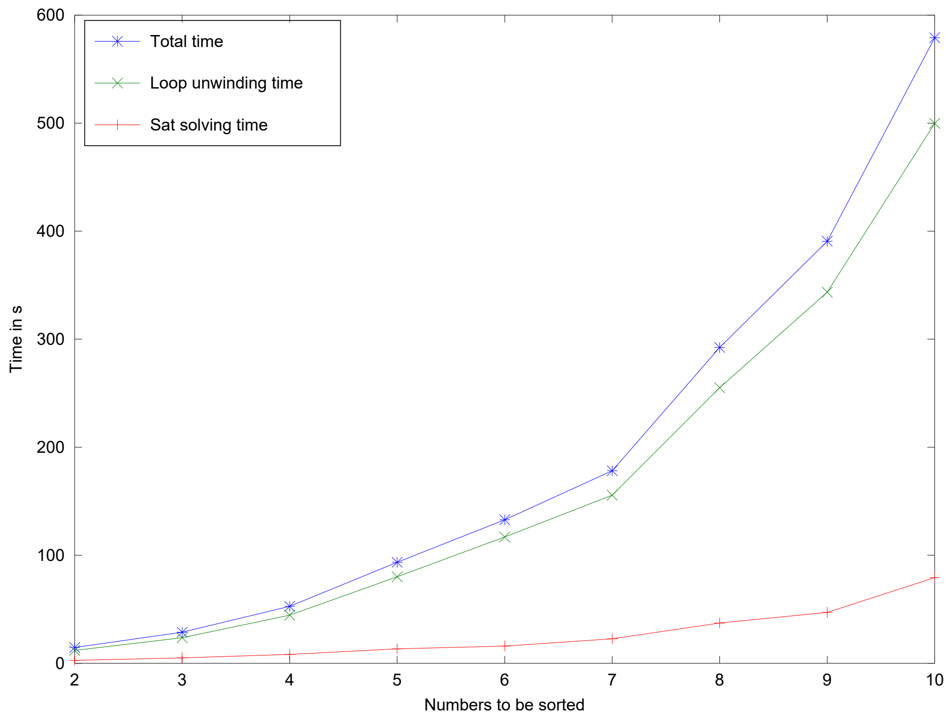


Figure 6.10: Verification times split into SAT solving and loop unwinding, increasing the array of to be sorted numbers n for *Bubble Sort*, with PIM = 2.

application does not rely on the event timer system and is not dependent on interrupts only POR with a maximum of 2 interrupts and PIM with an interrupt distance of 2 statements was used.

In Table 6.5 the verification results are summarized. In total 680 assertions were checked, of these assertions, 19 assertions are part of the LCD driver and 3 are part of the actual application, which makes sure that the application algorithm is correct. The rest of the assertions are the automatically generated assertions for pointer check and division by zero. The result table shows results for sorting 5 and 10 numbers and the corresponding verification problem sizes. A graphical comparison of the verification times for increasing numbers to be sorted is shown in Figure 6.8. The verification times and problem sizes show that an exponential growth in verification time and problem size occurs and that POR is faster than PIM. In Figure 6.9 and Figure 6.10 the run times are shown split into preprocessing and SAT solving times. For PIM most of the time is spent in the preprocessing phase. Furthermore, preprocessing time does not increase linear for PIM but rather exponentially similar to the expected overall SAT solving behavior.

Comparing the pure SAT solving times and formula sizes it seems that the formula generated when using PIM is easier to solve than for POR. In the case of $n = 5$ and $PIM = 2$ the formula is larger than for $n = 10$ and $POR = 2$, however, SAT solving is much faster.

6.6 *3-axis Acceleration Sensor with Rotation Detection* Application

This example is the most complex of the evaluated applications, taken from a real-world embedded system, which can be used for fall detection. It uses a 3-axis acceleration sensor to sample acceleration values. The purpose of this application is to signal that a certain rotation on the x-axis of the system has been detected. When this happens an LED of the system is enabled. The source code used for this application is shown in Appendix A.1.6 and A.1.7.

For this application, the *Contiki* sensor subsystem was used, which collects information from registered sensors and sends events to application processes, when new sensor data is available. This centralized data collection allows it for several applications to use the sensor data, which only needs to be collected once. To use a sensor with the subsystem it has to be registered and the data collection has to be implemented. For the used acceleration sensor the registration within the *Contiki* sensor system is shown exemplarily in Appendix A.1.7. Three functions must be provided to implement the

registration:

- A configuration function (`configure_acc` lines 36-60). This function is called automatically when the sensor is configured to be used. The pre-defined types `SENSORS_HW_INIT` and `SENSORS_ACTIVE` are passed as a parameter for initialization and activation of the sensor. When the sensor is activated and not yet running (lines 45-49), a process is started (`acc_update_process` lines 17-34), which periodically pulls data from the sensor using the event timer system. When new data is available an event is sent as a *ProcessBroadcast* to all processes in the system using the function `sensors_changed` (line 30).
- A value access function (`value_acc` lines 68-83). This function is used to retrieve sensor values captured from the sensor. In the case of the acceleration sensor, these are values for each axis, which are stored in the array `value`. The modeling of this sensor has been described exemplarily in Figure 4.5(b) of Section 4.3.
- A status function (`status_acc` lines 88-96). This function returns the current status of the sensor (active or inactive)

These functions are registered within the system using the macro `SENSORS_SENSOR` (line 99).

The source code of the main application process, which uses the sensor is shown in Appendix A.1.6. The basic idea of the application is to detect a rotation of the embedded system which includes the sensor. Therefore, the last two retrieved sensor values are stored in the variables `xyz1` and `xyz2` (lines 33-34). As a history of two sensor values is not sufficient to detect a rotation, the array `buffer` (line 37) is used as history that stores, whether before retrieved acceleration values were over a certain threshold.

During processing, the application first stores the last received acceleration value in the variable `xyz2` using the **C** `memcpy` function[23] (line 53). If an event from the acceleration sensor is received (lines 55-59), new acceleration values are retrieved from the sensor using the function `get_xyz`. Using these values the rotation detection algorithm then performs its calculation (lines 71-106). The principal idea is that each time the difference between current and last acceleration value exceeds the value 20 in the x-direction, in the `buffer` array the variable 1 is set. When the sum of the first 5 buffer values equals 5 the LED is turned on, when it is 0 the LED is turned off. Using this filtering the rotation detection gets more robust.

Verification results Results for the verification of automatically generated assertions are shown in in Table 6.6. The results show verification times for one main loop

[23]The `memcpy` function is available in the built-in CBMC **C** library.

lines of code	596	
number of assertions	836	
main loop unwinding	1	
interrupt modeling style	POR	PIM
max. number of timer interrupts	10	-
timer interrupt period	-	200
program expression steps	337,093	1,168,477
variables	7,677,005	13,781,360
clauses	27,683,272	46,443,662
unwinding and preprocessing in seconds	48,425.82	57,995.04
SAT solving in seconds	2,424.74	526.11

Table 6.6: Verification time and problem size for the *3-axis acceleration* example verifying automatically generated assertions.

counterexample size in statements	4,271
interrupt modeling style	PIM
timer interrupt period	200
program expression steps	4,737,759
variables	73,468,255
clauses	278,320,484
unwinding and preprocessing in seconds	274,471.96
SAT solving in seconds	13,012.50

Table 6.7: Verification time for generating a minimal test case for the *3-axis acceleration* example to turn off the LED of the embedded system.

unwinding. For POR with 10 interrupts verification takes about 14 hours and for PIM over 16 hours. With each further unwinding also of internal loops the verification times further increase significantly, therefore making a complete main loop unwinding of 2 unfeasible for this example, verification did not terminate after several days of running. A main loop unwinding of 1 means, that 1 event from the kernel event queue is executed[24].

Test case generation Using this example, the application of BMC for minimal test case generation can be demonstrated. Test cases generation is performed as described in Section 5.3, the resulting verification times for PIM are shown in Table 6.7. The test case that was generated turns the LED of the embedded system off (line 99 in Appendix A.1.6). To generate this test case, an assertion was added at this code line, which must not be reached (`assert(0)`). CBMC was able to generate a counterexample trace with a size of 4271 steps within the **C** program, which can violate this assertion. As loops are incremented individually, the resulting counterexample is

[24] All processes of the system are however run at least once using the `AUTOSTART_PROCESSES` macro. Furthermore, processes for the event timer system and the sensor subsystem are started automatically and also run at least once.

minimal. However, the runtimes for generating this counterexample are very high, the program needs to be further unwound then for checking the automatically generated assertions, leading to a larger size of the SAT formula and consequently to higher verification times.

6.7 Summary and Result Discussion

The presented verification results demonstrate that it is possible to verify realistic *Contiki* applications using the described verification approach. It could was demonstrated that it is possible to use driver models of actual hardware to verify unmodified *Contiki* applications on the implemented verification platform. Furthermore, it was shown that it is possible to find violated assertions and to generate test cases. Moreover, using the verification platform the differences between the PIM and POR interrupt modeling styles were demonstrated regarding their capability for finding bugs. The verification of *application-dependent assertions*, which can be placed in the application and *platform-dependent assertions*, which are part of the drivers used for verification (and are application independent), was successfully shown using the *Bubble sort* application.

The results show that no unexpected bugs were found in the verified applications or in the *Contiki* kernel. This is expected as the *Contiki* kernel is very stable and used in many devices. Comparing the verification times for the checked applications, although similar in size regarding lines of code, differ in verification complexity depending on the used number of timers and events created. For example, the *Hello World* and *LED Blink* application are similar in lines of code, but verification times differ significantly. Overall, the verification times vary from several seconds for the *Hello World* example and can go up to days for the more complex applications, especially for larger main loop unwindings. The main determining factor for verification time is the number of main loop unwindings, which limits the search depth by restricting the number of events checked for the system. Furthermore, the chosen approach to unroll all application loops and to only limit the main loop unwindings in the kernel could successfully be applied.

The comparison of the two approaches for interrupt modeling, the in this work developed PIM approach with the existing POR approach, highlighted several key differences. PIM allows it to prove assertions, which depend on the number of interrupt occurrences, which is important for *Contiki* and its event-driven approach. For example, when verifying the *Blink* application with an event queue overflow, POR leads due to the over-abstraction of the interrupt behavior to *false positive* counterexamples, which do not exist on real hardware. When using PIM this application can be proven.

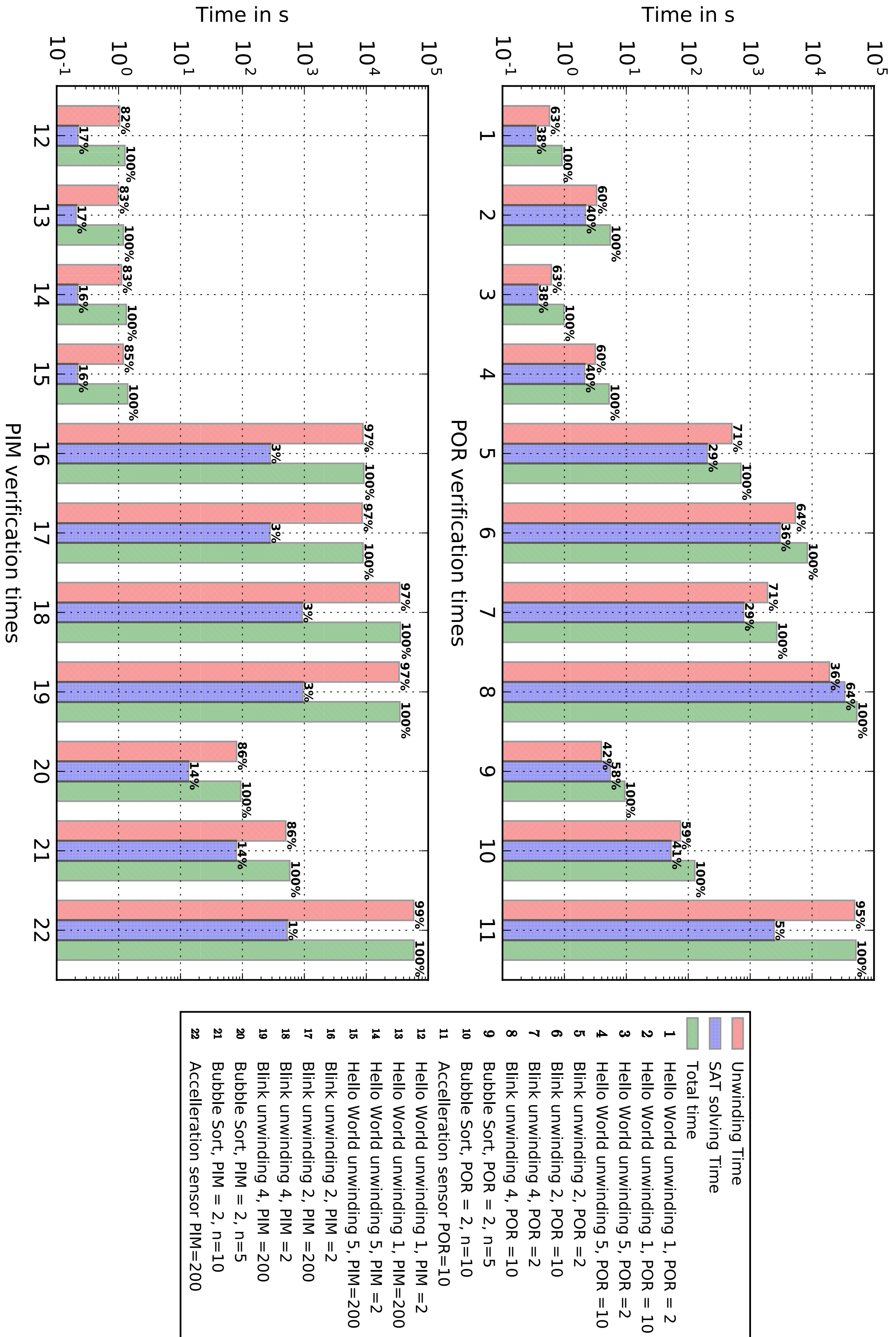


Figure 6.11: Overall comparison of POR and PIM verification results of examples in Section 6.2 - 6.6.

Another application which could only be verified using PIM is the *LED Fader* application, which uses busy waiting. When using POR there is always a program path where no interrupt is triggered, leading to an endless loop when unwinding, so that an unwinding assertion is always violated.

Furthermore, for the generation of test cases, PIM is better suited as it leads to more realistic test cases as when using POR as shown by the *acceleration sensor* application.

The more accurate modeling of interrupt behavior of PIM leads however to longer verification times compared with POR. In Figure 6.11 a summary is shown, which compares the runtimes for PIM with POR for the run applications[25]. In all applications, it can be seen that verification times for POR increases with the number of considered interrupt occurrences, whereas the time and also verification problem size (variables and clauses) is constant for PIM independent of the chosen event timer interrupt period. POR is faster for a small number of interrupts, however, when increasing the number of interrupts, unwinding and preprocessing and SAT solving times increase significantly and POR can become slower than PIM. It shows that POR is best suited when the system is checked for only a few possible interrupt occurrences. However, the number of sufficient interrupt occurrences has to be manually determined.

Another significant difference is, that for PIM more time is spent in unwinding the program and creating the SAT formula. However, the larger programs after transformation and unwinding (about 3-4 times larger) do not lead to an increase in runtime in this order of magnitude. In Figure 6.12 the verification time is plotted over the program size. It shows that the programs are getting much bigger for PIM, however, runtimes are in the same range as for POR.

Overall it could be demonstrated that verification is possible for the chosen programs, with the limiting factor being the unwindings of the main loop. The PIM approach has the advantage of being more accurate and therefore can find more bugs, while reducing the number of *false positives*. Differences in verification time depend on the number of interrupts for POR.

[25] Only the runtimes are listed for applications which could be verified both with PIM and POR.

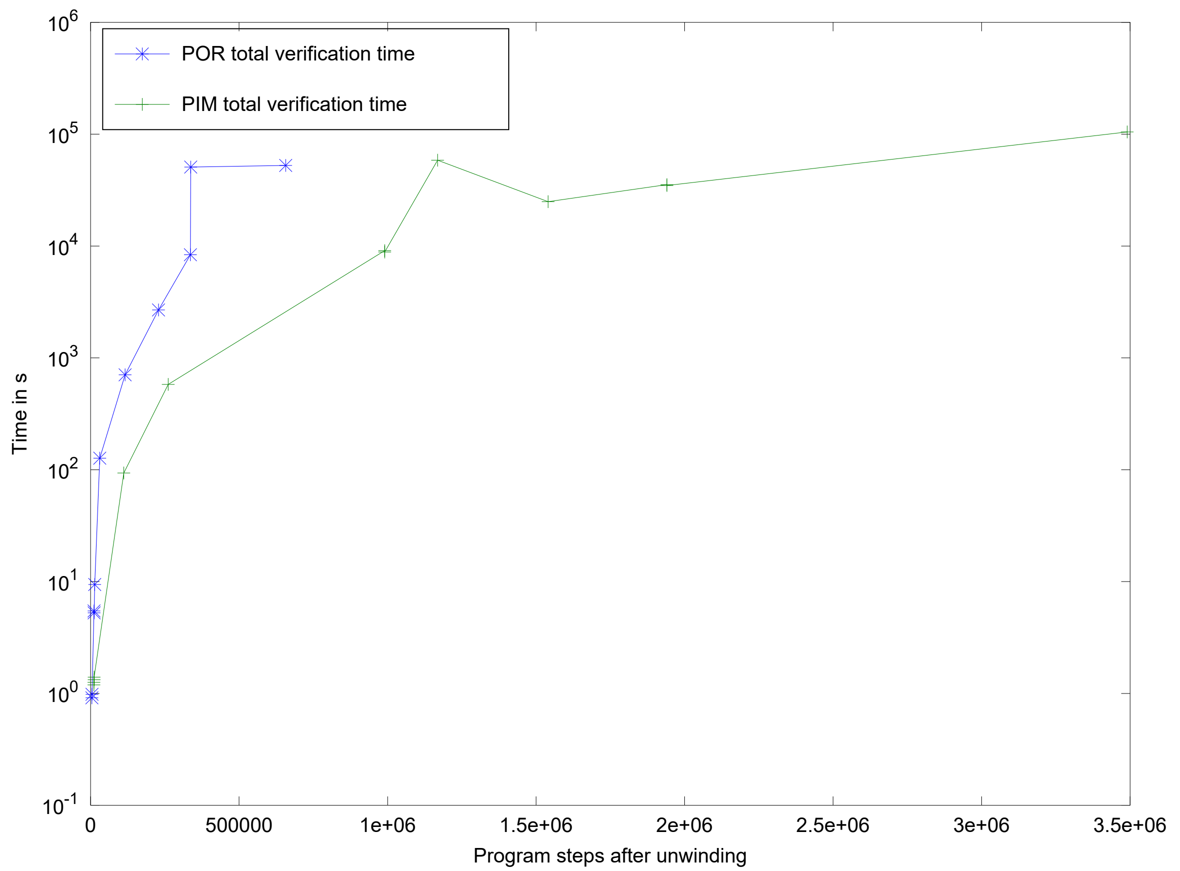


Figure 6.12: Influence of program steps on verification times for POR and PIM, based on examples in Section 6.2 - 6.6.

Chapter 7
Conclusions

Formal verification is a challenge considering computation resources, but also modeling and specification requires high effort. Nowadays, advancements in verification tools and computer processing speed allow it to verify software with techniques like *model checking*, originally developed for the verification of digital hardware.

In this thesis, it was shown exemplarily using software for the operating system *Contiki* that it is possible to apply *bounded model checking* to verify software of embedded systems. Thereby, one challenge was to find the correct level of abstraction for modeling the embedded system hardware on which the applications runs. A too detailed model cannot be handled by the verification tools, due to a too large state space. A too coarse model leads to unrealistic software behavior and to counterexamples during verification that cannot occur on the real system. Therefore, hardware modeling was performed at the level of the *Contiki* drivers, compared to other approaches. These approaches performed modeling at the level of hardware registers of the processor, on which the software is executed. By modeling the hardware using non-deterministic variables, the software is verified for all possible input combinations, compared to simulation-based approaches.

It could be shown that it is possible to automatically verify unmodified *Contiki* applications by this method. As bounded model checking was used, loops in the source code must be limited. In the chosen approach, the *Contiki* main scheduler loop was limited, making it possible to control the size of the verification problem and to find a trade-off between verification depth and verification runtime.

An important aspect of software for embedded systems is the correct representation of interrupt behavior, which has a direct effect on the program flow of an application. It was shown that current interrupt modeling methods are not suited for systems that rely on the use of periodically occurring interrupts. Therefore, in this thesis, a new technique called *periodic interrupt modeling* (PIM) was introduced. In comparison

with existing approaches based on the *interleaving model* and *partial order reduction*, it is now possible to verify timing related properties of embedded system software. PIM can be used independently of the verification of *Contiki* applications for all applications that rely on periodic interrupts. When applying PIM the runtime of an application is taken into account and thereby the number of unrealistic counterexamples is reduced during verification. This also allows it to use the verification approach for the generation of test cases. Furthermore, a higher level of verification automation is possible, as applications that rely on busy waiting can be verified without modifications.

The higher level of modeling accuracy introduced by periodic interrupt modeling increases the size of the program and therefore leads to a higher verification effort. However, through experiments based on *Contiki* applications, it could be shown that the increase in program size doesn't increase verification time in the same manner. Periodic interrupt modeling is, therefore, a viable method for the verification of embedded system software, which allows it to verify a new class of applications, based on periodic occurring interrupts.

In this thesis, verification was limited to applications with only one interrupt. Future research should examine the influence of multiple interrupts in a system, combining periodic and non-periodic interrupts as well as the modeling of interrupt priorities. First research results in this direction are presented in [KLM+15]. However, in their paper the problems when verifying timing related properties, as discussed in this thesis are not examined.

Another important topic for the verification of embedded system software is searching the complete state space when applying model checking. Therefore, new interpolation based model checking algorithms like IC3 [Bra11] have been developed, which extend SAT-based bounded model checking (BMC). In contrast to BMC, the complete state space is examined without giving a loop unwinding bound. An overview and a discussion of these techniques is given in [GOY14]. Applying these algorithms for unbounded software as *Contiki* systems is also a research area for further examinations.

Although model checking algorithms and SAT solving techniques have improved, the size of the state space is often too large to handle for formal tools. In the domain of verification of digital circuits, verification methodologies like the *universal verification methodology* (UVM) [Acc14] combine simulation with randomly generated input values. Furthermore, *functional coverage* is used to measure the verification progress. Extending this approach to also verify the correctness of software running on an embedded system hardware, when model checking is not anymore feasible, is also an interesting topic for future research.

Appendix A

A Appendix

A.1 Example Applications Used for Evaluation

A.1.1 Transformed Source Code of Hello World Application for POR

```
1  static char process_thread_hello_world_process(struct pt *process_pt,
2          unsigned char ev, void *data)
3  {
4    char PT_YIELD_FLAG = (char)1;
5    if((signed int)process_pt->lc == 0)
6      printf("Hello, world\n");
7
8    PT_YIELD_FLAG = (char)0;
9    process_pt->lc = (unsigned short int)0;
10   return (char)3;
11 }
```

A.1.2 Transformed Source Code of Hello World Application for PIM

```
1  static char process_thread_hello_world_process(struct pt *process_pt,
2    unsigned char ev, void *data)
3  {
4    char PT_YIELD_FLAG = (char) 1;
5    periodic_interrupt();
6    if (((signed int) process_pt->lc) == 0)
7    {
8      periodic_interrupt();
```

```
 9       printf("Hello, world\n");
10     }
11
12     periodic_interrupt();
13     PT_YIELD_FLAG = (char) 0;
14     periodic_interrupt();
15     process_pt->lc = (unsigned short int) 0;
16     periodic_interrupt();
17     return (char) 3;
18   }
```

A.1.3 Source Code of *LED Blink* Application Possibly Triggering an Event Queue Overflow

```
 1   PROCESS(blink_process, "Blink");
 2   AUTOSTART_PROCESSES(&blink_process);
 3   PROCESS_THREAD(blink_process, ev, data)
 4   {
 5           PROCESS_EXITHANDLER(goto exit;)
 6     PROCESS_BEGIN();
 7     static struct etimer et1;
 8     static struct etimer et2;
 9     static struct etimer et3;
10     static struct etimer et4;
11     while(1) {
12       etimer_set(&et1, CLOCK_SECOND);
13       etimer_set(&et2, 2*CLOCK_SECOND);
14       etimer_set(&et3, 3*CLOCK_SECOND);
15       etimer_set(&et4, 4*CLOCK_SECOND);
16       PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et1));
17       leds_on(LEDS_ALL);
18       PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et2));
19       leds_off(LEDS_ALL);
20       PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et3));
21       leds_on(LEDS_ALL);
22       PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et4));
23       leds_off(LEDS_ALL);
24     }
25     exit:
26       leds_off(LEDS_ALL);
27       PROCESS_END();
28   }
```

A.1.4 Source Code of LED Fader Application

```
1  #include "contiki.h"
2  #include "dev/leds.h"
3
4  PROCESS(fader_process, "LED fader");
5  AUTOSTART_PROCESSES(&fader_process);
6
7  #define ON 1
8  #define OFF 0
9
10 struct fader {
11   struct pt fade_pt, fade_in_pt, fade_out_pt;
12   struct etimer etimer;
13   int led;
14   int delay;
15 };
16
17 static unsigned char onoroff;
18
19 static
20 PT_THREAD(fade_in(struct fader *f))
21 {
22   PT_BEGIN(&f->fade_in_pt);
23
24   for(f->delay = 9; f->delay > 1; f->delay -= 1) {
25
26     printf("Fading In \n");
27     leds_on(f->led);
28     clock_delay(10 - f->delay);
29     leds_off(f->led);
30     clock_delay(f->delay);
31     PT_YIELD(&f->fade_in_pt);
32   }
33
34   PT_END(&f->fade_in_pt);
35 }
36
37 static
38 PT_THREAD(fade_out(struct fader *f))
39 {
40   PT_BEGIN(&f->fade_out_pt);
41
42   for(f->delay = 1; f->delay < 9; f->delay += 1) {
43
44     printf("Fading Out \n");
45     leds_on(f->led);
46     clock_delay(10 - f->delay);
```

```
47       leds_off(f->led);
48       clock_delay(f->delay);
49       PT_YIELD(&f->fade_out_pt);
50     }
51
52     PT_END(&f->fade_out_pt);
53   }
54
55   static
56   PT_THREAD(fade(struct fader *f))
57   {
58     PT_BEGIN(&f->fade_pt);
59
60     while(1) {
61       printf("Fading \n");
62       PT_SPAWN(&f->fade_pt, &f->fade_in_pt, fade_in(f));
63       PT_SPAWN(&f->fade_pt, &f->fade_out_pt, fade_out(f));
64
65       etimer_set(&f->etimer, CLOCK_SECOND * 4);
66       PT_WAIT_UNTIL(&f->fade_pt, etimer_expired(&f->etimer));
67     }
68
69     PT_END(&f->fade_pt);
70   }
71
72   static void
73   init_fader(struct fader *f, int led)
74   {
75     PT_INIT(&f->fade_pt);
76     PT_INIT(&f->fade_in_pt);
77     PT_INIT(&f->fade_out_pt);
78     f->led = led;
79   }
80
81   PROCESS_THREAD(fader_process, ev, data)
82   {
83     static struct fader red;
84     static struct timer timer;
85     static struct etimer etimer;
86
87     PROCESS_BEGIN();
88
89     init_fader(&red, LEDS_RED);
90
91     printf("Scheduling red\n");
92     timer_set(&timer, CLOCK_SECOND);
93     while(!timer_expired(&timer)) {
94       printf("Scheduling red, waiting for timer\n");
```

```
95      PT_SCHEDULE(fade(&red));
96    }
97
98    etimer_set(&etimer, CLOCK_SECOND * 4);
99    fader_on();
100
101   while(1) {
102     printf("Main Loop\n");
103     PROCESS_WAIT_EVENT();
104
105     if(ev == PROCESS_EVENT_TIMER) {
106       etimer_set(&etimer, CLOCK_SECOND * 4);
107       process_poll(&fader_process);
108     }
109
110     if(onoroff == ON &&
111          PT_SCHEDULE(fade(&red))) {
112       process_poll(&fader_process);
113     }
114   }
115   PROCESS_END();
116 }
117
118 void fader_on(void)
119 {
120   onoroff = ON;
121   process_poll(&fader_process);
122 }
123
124 void fader_off(void)
125 {
126   onoroff = OFF;
127 }
```

A.1.5 Source Code of Bubble Sort with LCD Application

```
1   #include "contiki.h"
2   #include "dev/lcd.h"
3   #define BUBBLE_SORT_NMAX 5
4
5   // Takes a number and displays it on the display
6   void printInteger(uint8_t n)
7   {
8     uint8_t temp;
9     uint8_t idx = 0;
10    lcd_clr();
```

```
11     while(n >= 10) {
12       temp = n%10;
13       #ifdef VERIFY
14       assert(temp < 10);
15       #endif
16       n = (n-temp)/10;
17       lcd_disp_char(idx, temp);
18       idx++;
19     }
20     #ifdef VERIFY
21     assert(n < 10);
22     #endif
23     lcd_disp_char(idx, n);
24     idx++;
25   }
26   uint8_t getIntArray(uint8_t a[], const uint8_t values[], uint8_t nmax)
27   {
28     uint8_t n = 0;
29
30     while (n < nmax) {
31       #ifdef SIMULATION
32       a[n] = values[n];
33       #endif
34       #ifdef VERIFY
35       a[n] = nondet_char();
36       #endif
37       n++;
38     }
39     return n;
40   }
41
42
43   void bubbleSort(uint8_t a[], uint8_t n)
44   /* It sorts in non-decreasing order the first N positions of A. It uses
45    * the bubble sort method.
46    */
47   {
48     uint8_t lcv;
49     uint8_t limit = n-1;
50     uint8_t temp;
51     uint8_t lastChange;
52     while (limit) {
53       lastChange = 0;
54       for (lcv=0;lcv<limit;lcv++)
55         if (a[lcv]>a[lcv+1]) {
56     temp = a[lcv];
57       a[lcv] = a[lcv+1];
58       a[lcv+1] = temp;
```

```
59          lastChange = lcv;
60            }
61          limit = lastChange;
62        }
63    }
64
65    #ifdef VERIFY
66    void checkResultArray(uint8_t a[], uint8_t n)
67    /* n is the number of elements in the array a.
68     * Check whether array a is sorted*/
69    {
70      uint8_t i;
71      for (i=0; i<n−1; ){
72        assert (a[i] <= a[i+1]);
73        i=i+1;
74      }
75    }
76    #endif
77
78    PROCESS(lcd_sort_process, ”LCD sort process”);
79    AUTOSTART_PROCESSES(&lcd_sort_process);
80
81    PROCESS_THREAD(lcd_sort_process, ev, data)
82    {
83      #ifdef SIMULATION
84      const uint8_t v[BUBBLE_SORT_NMAX] = {41,0,1,42,15};
85      #else
86      const uint8_t v[BUBBLE_SORT_NMAX];
87      #endif
88      static uint8_t x[BUBBLE_SORT_NMAX];
89      static uint8_t hmny;
90      static uint8_t i;
91      PROCESS_BEGIN();
92
93      lcd_init();
94      lcd_disp_all_segs();
95      lcd_clr();
96
97      // get integer values to be sorted
98      hmny = getIntArray(x,v, BUBBLE_SORT_NMAX);
99      // Print not sorted array
100     for (i=0; i<BUBBLE_SORT_NMAX; ){
101       printInteger(x[i]);
102       i++;
103     }
104     //Clear Display
105     lcd_disp_all_segs();
106     lcd_clr();
```

```
107
108   /* Sort the array */
109   bubbleSort(x,hmny);
110   #ifdef VERIFY
111   checkResultArray(x,hmny);
112   #endif
113
114   /* Print sorted array */
115   for (i=0; i<BUBBLE_SORT_NMAX; ){
116     printInteger(x[i]);
117     i++;
118   }
119   PROCESS_END();
120 }
```

A.1.6 Source Code of 3-axis Acceleration Sensor with Rotation Detection Application

```
1  #include "contiki.h"
2  #include "dev/leds.h"
3  #include "dev/acc_sensor.h"
4  #include "lib/sensors.h"
5
6  #include "sys/clock.h"
7
8  #include "dev/watchdog.h"
9
10 #include "contiki-conf.h"
11 #include <stdio.h> /* For printf() */
12 #include <string.h> /* For memcpy() */
13
14 PROCESS(acc_test_process, "Accelerometer test process");
15 AUTOSTART_PROCESSES(&acc_test_process);
16
17 /*
18  * Helper function to get a xyz-triple with matching values
19  * (in order to avoid additional statements or breaks between two single
20  * value-function calls, i.e. acc->value(XAXIS))
21  */
22 static void get_xyz(struct sensors_sensor *acc, int16_t *xyz)
23 {
24   xyz[0] = (int16_t) acc->value(XAXIS);
25   xyz[1] = (int16_t) acc->value(YAXIS);
26   xyz[2] = (int16_t) acc->value(ZAXIS);
27 }
```

```
28
29 PROCESS_THREAD(acc_test_process, ev, data)
30 {
31   struct sensors_sensor *acc;
32   int i;
33   static int16_t xyz1[3] = {0,0,0};
34   static int16_t xyz2[3] = {0,0,0};
35
36   static int buf_counter;
37   static int buffer[10];
38   static long int state;
39
40
41   PROCESS_BEGIN();
42
43   /* Variable init. */
44   state = 0;
45   buf_counter = 0;
46   for(i = 0;i<10; i++) {
47     buffer[i] = 0;
48   }
49
50   while(1)
51   {
52
53     memcpy(xyz2, xyz1, 6); //copy 6 bytes
54
55     PROCESS_WAIT_EVENT_UNTIL(ev == sensors_event);
56     if(data == &acc_sensor) {
57       acc = (struct sensors_sensor *)data;
58       get_xyz(acc, xyz1);
59     }
60
61     printf("x1=%4d y1=%4d z1=%4d\r\n", xyz1[0] , xyz1[1], xyz1[2]);
62     printf("x2=%4d y2=%4d z2=%4d \r\n", xyz2[0] , xyz2[1], xyz2[2]);
63     printf("x=%4d y=%4d z=%4d\r\n", xyz2[0]-xyz1[0] , xyz2[1]-xyz1[1], xyz2[2]-xyz1[2]);
64     printf("state: %d\r\n", state);
65
66     /*
67      * Here starts a simple rotation detection sensitive to the x-axis
68      * Turning right --> red LED on
69      * Turning left --> red LED off
70      */
71     if( xyz2[0]-xyz1[0] < -20 || xyz2[0]-xyz1[0] > 20)
72     {
73       if (state > 13)
74       {
75         state = 0;
```

```
76            buf_counter = 0;
77          }
78
79          if (buf_counter < 10) {
80            if ( (xyz2[0]-xyz1[0]) > 0)
81            {
82              buffer[buf_counter++] = 1;
83            }
84            else
85            {
86              buffer[buf_counter++] = 0;
87            }
88          }
89          if (buf_counter <= 5)
90          {
91            printf("%d: %d%d%d%d%d\r\n",buf_counter, buffer[0], buffer[1],
92            buffer[2], buffer[3], buffer[4] );
93
94            if ((buffer[0] + buffer[1] + buffer[2] + buffer[3] + buffer[4]) == 5) {
95              leds_on(LEDS_RED);
96            }
97
98            if ((buffer[0] + buffer[1] + buffer[2] + buffer[3] + buffer[4]) == 0) {
99              leds_off(LEDS_RED);
100           }
101         }
102       }
103       else
104       {
105         state++;
106       }
107     }
108
109     PROCESS_END();
110   }
```

A.1.7 Source Code 3-axis Acceleration Sensor Declaration for *Contiki*

```
1   #include "adxl345.h"
2   #include "acc_sensor.h"
3   #include "lib/sensors.h"
4   #include "contiki-conf.h"
5
6
```

```
7  const struct sensors_sensor acc_sensor;
8  static int active; /* 1-sensor on, 0-sensor off */
9  static int16_t xyz[3];
10
11 /*
12  * Process for updating the sensor's value in a defined interval
13  * (see "contiki_conf.h" for definition of ACC_UPDATE_TIME)
14  */
15 PROCESS(acc_update_process, "Acceleration Sensor Update Process");
16
17 PROCESS_THREAD(acc_update_process, ev, data)
18 {
19         static struct etimer update_timer;
20
21         PROCESS_BEGIN();
22
23         adxl345_get_xyz(xyz);
24         etimer_set(&update_timer,0.08*CLOCK_SECOND);
25
26         while(1) {
27                 PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&update_timer));
28                 /* update new sensor values */
29                 adxl345_get_xyz(xyz);
30                 sensors_changed(&acc_sensor);
31                 etimer_set(&update_timer, 0.08*CLOCK_SECOND);
32         }
33         PROCESS_END();
34 }
35
36 static int configure_acc(int type, int value)
37 {
38         switch(type) {
39           case SENSORS_HW_INIT:
40             active = 0;
41             adxl345_init_dev();
42             return 1;
43           case SENSORS_ACTIVE:
44             if(value) {
45               if(!active) {
46                 active = 1;
47                 process_start(&acc_update_process,NULL);
48                 adxl345_enable_measurement();
49               }
50             } else {
51                 if(active) {
52                         active = 0;
53                         process_exit(&acc_update_process);
54                         adxl345_disable_measurement();
```

```
55                    }
56                }
57                return 1;
58              }
59              return 0;
60    }
61
62    /*
63     * returns sensor value depending on type
64     * type: 0 - x-axis
65     *     1 - y-axis
66     *     2 - z-axis
67     */
68    static int value_acc(int type)
69    {
70            int16_t value;
71            switch(type) {
72            case XAXIS:
73                    value = xyz[0];
74                    break;
75            case YAXIS:
76                    value = xyz[1];
77                    break;
78            case ZAXIS:
79                    value = xyz[2];
80                    break;
81            }
82            return (int)value;
83    }
84
85    /*
86     * returns status of sensor (activated, deactivated)
87     */
88    static int status_acc(int type)
89    {
90            switch(type) {
91              case SENSORS_ACTIVE:
92              case SENSORS_READY:
93                return active;
94              }
95              return 0;
96    }
97
98    /* Macro for declaration of sensor struct */
99    SENSORS_SENSOR(acc_sensor, ACC_SENSOR_NAME, value_acc, configure_acc, status_acc);
```

A.2 *Contiki* Kernel Implementation Details

This appendix describes further details of the *Contiki* kernel, which have either derived from the source code or the documentation of the *Contiki* project.

A.2.1 *Contiki* Defined Kernel Events

The official *Contiki* wiki page [Con17c] defines the following Kernel events and gives the following descriptions. Further events can be added by the user as described in Section 3.3.

Event name	**Description**
PROCESS_EVENT_NONE	This event identifier is not used.
PROCESS_EVENT_INIT	This event is sent to new processes when they are initiated.
PROCESS_EVENT_POLL	This event is sent to a process that is being polled.
PROCESS_EVENT_EXIT	This event is sent to a process that is being killed by the kernel. The process may choose to clean up any allocated resources, as the process will not be invoked again after receiving this event.
PROCESS_EVENT_CONTINUE	This event is sent by the kernel to a process that is waiting in a PROCESS_YIELD() statement.
PROCESS_EVENT_MSG	This event is sent to a process that has received a communication message. It is typically used by the IP stack to inform a process that a message has arrived, but can also be used between processes as a generic event indicating that a message has arrived.
PROCESS_EVENT_EXITED	This event is sent to all processes when another process is about to exit. A pointer to the process control block of the process that is existing is sent along the event. When receiving this event, the receiving processes may clean up state that was allocated by the process that is about to exit.
PROCESS_EVENT_TIMER	This event is sent to a process when an event timer (etimer) has expired.

A.2.2 *Contiki* Defined Process States

A *Contiki* process can take the following states:

Process state	Description
PROCESS_STATE_NONE	This process is not running on the system.
PROCESS_STATE_RUNNING	This process is currently being executed on the system.
PROCESS_STATE_CALLED	This process has been called, but is not currently executed.

A.2.3 Process macros of *Contiki*

Contiki implements the following process macros which form a convenience layer around the original *protothreads* API described in [DSVA06], by adding the possibility to wait for events. These macros are described in [Con17c] as:

Process macro	Description
PROCESS_BEGIN	Declares the beginning of a process *protothread*.
PROCESS_END	Declares the end of a process *protothread*.
PROCESS_EXIT	Exit the process.
PROCESS_WAIT_EVENT	Wait for any event.
PROCESS_WAIT_EVENT_UNTIL	Wait for an event, but with a condition.
PROCESS_YIELD	Wait for any event, equivalent to PROCESS_WAIT_EVENT().
PROCESS_WAIT_UNTIL	Wait for a given condition, may not yield the process.
PROCESS_PAUSE	Temporarily yield the process.

A.2.4 *Protothread* Macros of *Contiki*

Furthermore, *Contiki* applications can use the original *protothreads* as described in [DSVA06]. A summary is as follows:

Process macro	**Description**
PT_BEGIN	Declare the start of a *protothread* inside the C function implementing the *protothread*.
PT_END	Declare the end of a *protothread*.
PT_EXIT	Exit the *protothread*. If the *protothread* was spawned by another *protothread*, the parent *protothread* will become unblocked and can continue to run.
PT_INIT	Initialize a *protothread*. Initialization must be done prior to starting to execute the *protothread*.
PT_THREAD	Declaration of a *protothread*.
PT_WAIT_UNTIL	Block and wait until condition is true.
PT_YIELD	Yield from the current *protothread*.
PT_SPAWN	Spawn a child *protothread* and wait until it exits.

A.2.5 *Contiki* Process API

Contiki processes are controlled using the following API, as described in the *Contiki* source code. These functions can be used in applications to control the behavior of processes.

function	**Description**
process_start	This function starts a process.
process_exit	This function can be used to exit a process.
process_post	This function posts an asynchronous event to a process.
process_post_synch	This function posts a synchronous event to a process.
process_is_running	This function checks whether a specified process is either in a running or called state in the system.

Used Abbreviations

API	Application Programming Interface
BMC	Bounded Model Checking
CTL	Computation Tree Logic
GCC	GNU Compiler Collection
HDL	Hardware Description Language
HW	Hardware
IoT	Internet of Things
ISR	Interrupt Service Routine
LTL	Linear Temporal Logic
POR	Partial Order Reduction
PIM	Periodic Interrupt Modeling
SW	Software
UVM	Universal Verification Methodology
WSN	Wireless Sensor Network

List of Figures

1.1 General system development and verification process applicable for embedded systems [BK08]. . . . 2

2.1 Wireless sensor network modeled using SystemC AMS [VPB+08]. . . . 11
2.2 A UVM testbench for the structured verification of a SystemC design [VKE+14]. . . . 12
2.3 Principle approach for counterexample guided abstraction refinement [DKW08]. . . . 18
2.4 Visualization of the LTL operators and their semantics. . . . 21
2.5 Kripke structure for a modulo 4 counter. . . . 22
2.6 Bounded paths without and with loop [BCC+03]. . . . 24
2.7 BMC principle for formulas of safety properties of the kind $\mathbf{G}f$. . . . 25
2.8 Loop unwinding as done for BMC and adding of unwinding assertions. 29
2.9 Example of translating a simplified program into a SAT formula [CKL04]. 29
2.10 Modulo 4 counterexample modeled for CBMC. . . . 31
2.11 Interleaving diamond for α and β [BK08, CGP00]. . . . 33
2.12 Two stuttering equivalent paths [CGP00]. . . . 34
2.13 Extended interleaving diamond with additional transition, illustrating Example 2.6. . . . 36

3.1 *Tmote sky* wireless sensor node as used for *Contiki* [TMo07]. . . . 40
3.2 Contiki example *LED Blink* application using *protothreads*. . . . 41
3.3 Contiki process communication in the *LED Blink* example. . . . 48
3.4 Implementation of the *protothread* macros `PROCESS_BEGIN` and `PROCESS_WAIT_EVENT_UNTIL`. . . . 50
3.5 Organization of the *Contiki* source code. . . . 52

4.1 Replacement of drivers and annotation of assertions for an abstract *Contiki* verification platform. . . . 57
4.2 Example assertions for the verification of *Contiki* based systems 60
4.3 *Contiki* LED API and implementations for hardware access and verification. . . . 62

4.3 *Contiki* LED API and platform implementations for hardware access and verification. 63
4.4 Cutout from memory card driver as used in *Contiki*. 63
4.5 Cutout from a 3-axis acceleration sensor driver as used in *Contiki*. . . 65
4.6 Cutout from the timer interrupt implementation of the *Contiki* event timer system. 68
4.6 Cutout from the timer interrupt implementation of the *Contiki* event timer system. 69
4.7 Example application that can be interrupted. 71
4.8 Example application instrumented with interrupt calls after each statement. 72
4.9 Example application after reduction of interrupt calls with POR. 73
4.10 Interrupt wrapper function used for POR. 74
4.11 Functions used for *periodic interrupt modeling*. 75
4.12 Example application transformed using PIM. 77

5.1 *Contiki* LED API and implementations for hardware access and verification. 81
5.2 Unwinding of loops and restriction of the overall main loop unwinding. 82
5.3 Restricting the number of possible interrupt calls for BMC when applying POR. 85
5.4 Verification framework for Contiki applications. 86

6.1 The *Hello World* application. 93
6.2 Verification times for increasing main loop unwinding for the *LED Blink* example, verifying automatically generated assertions for POR and PIM. 96
6.3 Verification times for increasing main loop unwinding for the *LED Blink* example, comparing automatically generated assertions with no assertions for POR and PIM. 98
6.4 Verification times split into SAT solving and loop unwinding for increasing main loop unwinding for the *LED Blink* example, with POR = 2. 99
6.5 Verification times split into SAT solving and loop unwinding for increasing main loop unwinding for the *LED Blink* example, with POR = 10. 99
6.6 Verification times split into SAT solving and loop unwinding for increasing main loop unwinding for the *LED Blink* example, with PIM = 2. 100

6.7 Verification times split into SAT solving and loop unwinding for increasing main loop unwinding for the *LED Blink* example, with PIM = 200. 100
6.8 Verification times with increasing array sizes for the *Bubble Sort* example, verifying assertions for POR and PIM. 106
6.9 Verification times split into SAT solving and loop unwinding, increasing the array of to be sorted numbers n for *Bubble Sort*, with POR = 2. . . . 107
6.10 Verification times split into SAT solving and loop unwinding, increasing the array of to be sorted numbers n for *Bubble Sort*, with PIM = 2. . . . 107
6.11 Overall comparison of POR and PIM verification results of examples in Section 6.2 - 6.6. 112
6.12 Influence of program steps on verification times for POR and PIM, based on examples in Section 6.2 - 6.6. 114

List of Tables

2.1 Overview on verification tools for software verification. 17

6.1 Verification times and problem sizes for *Hello World* example verifying automatically generated assertions. 95

6.2 Verification times and problem sizes for *LED Blink* example verifying automatically generated assertions. 97

6.3 Verification time and problem size for the *LED Blink* example with *event queue full assertion*. 102

6.4 Verification time and problem size for the *LED Fader* example verifying automatically generated assertions using PIM. 104

6.5 Verification time and problem size for the *Bubble Sort* example verifying automatically generated assertions and *application*- and *platform-dependent assertions*. 106

6.6 Verification time and problem size for the *3-axis acceleration* example verifying automatically generated assertions. 110

6.7 Verification time for generating a minimal test case for the *3-axis acceleration* example to turn off the LED of the embedded system. 110

List of Algorithms

3.1 *Contiki* kernel initialization and main loop. 44

3.2 Function `DoPoll`: Handling of poll-requests. 45

3.3 Function `CallProcess`: Invokes a process using an event. 45

3.4 Function `ExitProcess`: Exits a process and removes it from the process list. 45

5.1 Pseudocode algorithm for running CBMC and unwinding loops. 83

Bibliography

[Acc14] Accellera Systems Initiative. Universal Verification Methodology (UVM) 1.2 Class Reference, 2014. URL: `http://accellera.org/images/downloads/standards/uvm/UVM_Class_Reference_Manual_1.2.pdf` [last visited 2017-06-10].

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[ASSC02] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, 2002.

[AVR17] Atmel AVR Microcontrollers, 2017. URL: `http://www.atmel.com/products/avr/` [last visited 2017-06-10].

[Bäh10] Helmut Bähring. Anwendungsorientierte Mikroprozessoren: Mikrocontroller und Digitale Signalprozessoren. *Anwendungsorientierte Mikroprozessoren*, 2010.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.

[BCM+90] J. R. Burch, Edmund Clarke, K. L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10ˆ20 states and beyond. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 428–439, 1990.

[BDE+15] M. Barnasconi, M. Dietrich, K. Einwich, T. Vortler, R. Lucas, J. P. Chaput, F. Pecheux, Z. Wang, P. Cuenot, I. Neumann, and T. Nguyen. UVM-SystemC-AMS Framework for System-Level Verification and Validation of Automotive Use Cases. *Design & Test, IEEE*, PP(99):1, 2015.

[BHvM09] Armin. Biere, Marin. Heule, and Hans. van Maaren. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications.*

IOS Press, 2009.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, Cambridge, Mass, 2008.

[BK11a] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Ganesh Gopalakrishnan, and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[BK11b] Doina Bucur and Marta Kwiatkowska. On software verification for sensor nodes. *Journal of Systems and Software*, 84(10):1693–1707, 2011.

[BLR11] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with SLAM. *Communications of the ACM*, 54(7):68–76, 2011.

[BMB+01] B. Berard, P. McKenzie, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer Berlin Heidelberg, 2001.

[BPV14] Martin Barnasconi, Francois Pecheux, and Thilo Vörtler. Advancing system-level verification using UVM in SystemC. In *Design and Verification Conference (DVCon), 2014*, 2014. URL: `http://events.dvcon.org/2014/proceedings/papers/01_1.pdf` [last visited 2017-06-10].

[Bra11] Aaron R. Bradley. SAT-based model checking without unrolling. In *Lecture Notes in Computer Science*, pages 70–87. Springer Berlin Heidelberg, 2011. `doi:10.1007/978-3-642-18275-4_7`.

[CAA+11] Alessandro Cimatti, Alberto Griggio, Andrea Micheli, I. Narasamdya, and Marco Roveri. Kratos - A Software Model Checker for SystemC, 2011. URL: `https://es-static.fbk.eu/tools/kratos/kratosdoc/manual.pdf` [last visited 2017-06-10].

[CBM17] CBMC Website, 2017. URL: `http://www.cprover.org/cbmc` [last visited 2017-06-10].

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[CCF+09] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does Astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, volume 8, pages 209–224, 2008.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT, Cambridge and Mass., 2. print. edition, 2000.

[CK03] E. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *Design Automation Conference, 2003. Proceedings of the ASP-DAC 2003. Asia and South Pacific*, pages 308–311, 2003.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[CKSY05] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based Predicate Abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer Verlag, 2005.

[CNR13] Alessandro Cimatti, I. Narasamdya, and Manuel Roveri. Software Model Checking SystemC. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(5):774–787, 2013.

[Con17a] Contiki OS, July 2017. URL: `http://www.contiki-os.org/` [last visited 2017-06-10].

[Con17b] Contiki supported hardware platforms, 2017. URL: `http://www.contiki-os.org/hardware.html` [last visited 2017-06-10].

[Con17c] Contiki wiki, 2017. URL: `https://github.com/contiki-os/contiki/wiki/` [last visited 2017-06-10].

[CPA17] Getting Started with CPAchecker, 2017. URL: `https://github.com/sosy-lab/cpachecker/blob/trunk/README.md` [last visited 2017-06-10].

[DG05] Rolf Drechsler and Daniel Große. System level validation using formal techniques. *Computers and Digital Techniques, IEE Proceedings* -, 152(3):393–406, 2005.

[DGV04] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible

operating system for tiny networked sensors. *2004. 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, 2004.

[DHKR11] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software Verification Using k-Induction. In Eran Yahav, editor, *Static Analysis: 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, pages 351–368. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[DKW08] 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*, 27(7):1165–1178, 2008.

[DSVA06] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado and USA, 2006.

[EDF+07] Joakim Eriksson, Adam Dunkels, Niclas Finne, Fredrik Osterlind, and Thiemo Voigt. Mspsim–an extensible simulator for msp430-equipped sensor boards. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, page 27, 2007.

[EÖF+09] Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Robert Sauter, and Pedro José Marrón. *COOJA/MSPSim: interoperability testing for wireless sensor networks.* ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009.

[GCC17] GCC, the GNU Compiler Collection, 2017. URL: `https://gcc.gnu.org/` [last visited 2017-06-10].

[GLD10] Daniel Große, Hoang M. Le, and Rolf Drechsler. Proving Transaction and System-level Properties of Untimed SystemC TLM Designs. *ACM & IEEE International Conference on Formal Methods and Models for Codesign*, 2010.

[GOY14] Shoham Sharon Grumberg Orna and Vizel Yakir. Sat-based model checking: Interpolation, ic3, and beyond. *NATO Science for Peace and Security Series, D: Information and Communication Security*, 36(Software Systems Safety):17–41, 2014. `doi:10.3233/978-1-61499-385-8-17`.

[HDG+09] Jan Haase, Markus Damm, Johann Glaser, Javier Moreno, and Christoph

Grimm. *SystemC-based power simulation of wireless sensor networks.* IEEE, 2009.

[Her10] Paula Herber. *A framework for automated HW-SW Co-verification of systemC designs using timed automata.* Dissertation, Technische Universität Berlin, Berlin, 2010. URL: `http://nbn-resolving.de/urn:nbn:de:kobv:83-opus-26364` [last visited 2017-06-10].

[HLGD12] Finn Haedicke, Hoang M. Le, Daniel Grosse, and Rolf Drechsler. *CRAVE: An advanced constrained random verification environment for SystemC.* IEEE, 2012.

[Hol04] Gerard J. Holzmann. *The spin model checker: Primer and reference manual.* Addison-Wesley, Boston, ©2004.

[HT05] Ali Habibi and Sofiene Tahar. On the Transformation of SystemC to AsmL Using Abstract Interpretation: Proceedings of the First International Workshop on Abstract Interpretation of Object-oriented Languages (AIOOL 2005). *Electronic Notes in Theoretical Computer Science*, 131:39–49, 2005.

[HTV+13] Alex Horn, Michael Tautschnig, Celina Val, Lihao Liang, Tom Melham, Jim Grundy, and Daniel Kroening. Formal Co-Validation of Low-Level Hardware/Software Interfaces. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 121–128. IEEE, 2013.

[IEE09] IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pages c1–626, Jan 2009. `doi:10.1109/IEEESTD.2009.4772740`.

[IEE10] IEEE Standard for Property Specification Language (PSL). *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pages 1–182, April 2010. `doi:10.1109/IEEESTD.2010.5446004`.

[IEE11] IEEE Standard for the Functional Verification Language e. *IEEE Std 1647-2011 (Revision of IEEE Std 1647-2008)*, pages 1–495, Aug 2011. `doi:10.1109/IEEESTD.2011.6006495`.

[IEE12] IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, Jan 2012. `doi:10.1109/IEEESTD.2012.6134619`.

[IEE13] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2012 (Revision of IEEE Std 1800-2009)*, pages 1–1315, Feb 2013. `doi:10.1109/IEEESTD.2013.6469140`.

[IEE16] IEEE Standard for Standard SystemC(R) Analog/Mixed-Signal Extensions Language Reference Manual. *IEEE Std 1666.1-2016*, pages 1–236, April 2016. doi:10.1109/IEEESTD.2016.7448795.

[Jor04] Günter Jorke. *Rechnergestützter Entwurf digitaler Schaltungen: Schaltungssynthese mit VHDL.* Fachbuchverl. Leipzig im Carl-Hanser-Verl., München and Wien, 2004.

[JZD09] Miloš Jevtić, Nikola Zogović, and Goran Dimić. Evaluation of wireless sensor network simulators. In *Proceedings of the 17th Telecommunications Forum (TELFOR 2009), Belgrade, Serbia*, pages 1303–1306, 2009.

[KCY03] D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Technical Report CMU-CS-03-126*. 2003.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KLM+15] Daniel Kroening, Lihao Liang, Tom Melham, Peter Schrammel, and Michael Tautschnig. Effective verification of low-level software with nested interrupts. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2015*, pages 229–234, 2015.

[KS05] D. Kroening and N. Sharygina. *Formal verification of SystemC by automatic hardware/software partitioning.* IEEE Computer Society, 2005.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, 2004.

[LFCJ09] Lucas Cordeiro, Bernd Fischer, Huan Chen, and Joao Marques-Silva. Semiformal Verification of Embedded Software in Medical Devices Considering Stringent Hardware Constraints. *Embedded Software and Systems, Second International Conference on*, pages 396–403, 2009.

[LLWC03] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. *TOSSIM: accurate and scalable simulation of entire TinyOS applications.* ACM, 2003.

[LMP+05] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An Operating System for Sensor Networks. In Werner Weber, JanM. Rabaey, and Emile Aarts, editors, *Ambient Intelligence*, pages 115–148. Springer Berlin Heidelberg, 2005.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *Inter-*

national Journal on Software Tools for Technology Transfer, 1(1-2):134–152, 1997.

[Mos15] MOS Technology 6502 - Wikipedia, the free encyclopedia, November 2015. URL: `https://en.wikipedia.org/w/index.php?oldid=689307762` [last visited 2017-06-10].

[MRR03] Wolfgang Müller, Wolfgang Rosenstiel, and Jürgen Ruf. *SystemC: Methodologies and applications*. Kluwer Academic, Boston, 2003.

[MSP17] MSP430 Microcontroller, 2017. URL: `http://www.ti.com/430brochure` [last visited 2017-06-10].

[MVÖ+10] Luca Mottola, Thiemo Voigt, Fredrik Österlind, Joakim Eriksson, Luciano Baresi, and Carlo Ghezzi. Anquiro: enabling efficient static verification of sensor network software. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*, pages 32–37. ACM, Cape Town and South Africa, 2010.

[Nik17] Niklas Sörensson Niklas Eén. MiniSat Page, 2017. URL: `http://minisat.se/Main.html` [last visited 2017-06-10].

[NWE+07] Nathan Cooprider, Will Archer, Eric Eide, David Gay, and John Regehr. Efficient memory safety for TinyOS. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 205–218. ACM, Sydney and Australia, 2007.

[ODE+06] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-Level Sensor Network Simulation with COOJA. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pages 641–648, 2006.

[OHT04] Karim Oumalou, Ali Habibi, and Sofiène Tahar. *Design for verification of a PCI bus in SystemC*. IEEE, 2004.

[PK09] T. Paul and G. S. Kumar. Safe Contiki OS: Type and Memory Safety for Contiki OS. *2009. ARTCom '09. International Conference on Advances in Recent Technologies in Communication and Computing*, pages 169–171, 2009.

[RDH03] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework: Bogor: an extensible and highly-modular software model checking framework. *ACM SIGSOFT Software Engineering Notes*, 28(5):267–276, 2003.

[RUPP12] Venkatesh R., Shrotri Ulka, Darke Priyanka, and Bokil Prasad. Test generation for large automotive models. In *Industrial Technology (ICIT), 2012 IEEE International Conference on*, pages 662–667, 2012.

[SBSV96] P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9):1167–1176, 1996.

[Sem14] N.X.P. Semiconductors. I^{2}C-bus specification and user manual: Rev. 6 — 4 April 2014. 2014. URL: `http://www.nxp.com/documents/user_manual/UM10204.pdf` [last visited 2017-06-10].

[Sim00] Simon Tatham. Coroutines in C, 2000. URL: `http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html` [last visited 2017-06-10].

[SMV17] The SMV System, 2017. URL: `http://www.cs.cmu.edu/~modelcheck/smv.html` [last visited 2017-06-10].

[SNBB11] Bastian Schlich, Thomas Noll, Jörg Brauer, and Lucas Brutschy. Reduction of Interrupt Handler Executions for Model Checking Embedded Software. In Kedar Namjoshi, Andreas Zeller, and Avi Ziv, editors, *Hardware and Software: Verification and Testing*, volume 6405 of *Lecture Notes in Computer Science*, pages 5–20. Springer Berlin / Heidelberg, 2011.

[SPI16] Serial Peripheral Interface Bus, October 2016. URL: `https://en.wikipedia.org/w/index.php?oldid=744586839` [last visited 2017-06-10].

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Staalmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, FMCAD '00, pages 108–125, London, UK, 2000. Springer-Verlag.

[TMo07] Data Sheet: tmote sky - Ultra low power IEEE 802.15.4 compliantwireless sensor module, 2007. URL: `http://www.snm.ethz.ch/snmwiki/pub/uploads/Projects/tmote_sky_datasheet.pdf` [last visited 2017-06-10].

[Var01] András Varga. The OMNeT++ discrete event simulation system. In *Proceedings of the European simulation multiconference (ESM'2001)*, volume 9, page 65, 2001.

[Var07] M. Y. Vardi. Formal Techniques for SystemC Verification; Position Paper. In *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pages 188–192, 2007.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[VK12] Kostyantyn Vorobyov and Padmanabhan Krishnan. Combining Static Analysis and Constraint Solving for Automatic Test Case Generation. In *2012 IEEE Fifth International Conference on Software Testing, Verifica-*

tion and Validation (ICST), pages 915–920, 2012.

[VKE+14] Thilo Vörtler, Thomas Klotz, Karsten Einwich, Yao Li, ZHi Wang, Marie-Minerve Louërat, Jean-Paul Chaput, François Pêcheux, Ramy Iskander, and Martin Barnasconi. Enriching UVM in SystemC with AMS extensions for randomization and functional coverage. In *Design and Verification Conference Europe (DVCon Europe), 2014*, 2014.

[VPB+08] Michel Vasilevski, Francois Pecheux, Nicolas Beilleau, Hassan Aboushady, and Karsten Einwich. Modeling refining heterogeneous systems with systemc-ams: application to wsn. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE), 2008*, pages 134–139. 2008.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *1st Symposium in Logic in Computer Science (LICS)*, pages 322–331, 1986.

[Wir14] Out in the Open: The Little-Known Open Source OS That Rules the Internet of Things, June 2014. URL: `http://www.wired.com/2014/06/contiki/` [last visited 2017-06-10].