Diss. ETH No. 19230

Testing of Wireless Sensor Networks

A dissertation submitted to the

ETH ZURICH

for the degree of

Doctor of Sciences

presented by

MATTHIAS WOEHRLE
Diplom in Electrical Engineering from TU Karlsruhe

born October 27, 1978
citizen of Germany

accepted on the recommendation of

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Koen Langendoen, co-examiner

2010

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. 114

MATTHIAS WOEHRLE

Testing of Wireless Sensor Networks

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A dissertation submitted to the
Swiss Federal Institute of Technology (ETH) Zürich
for the degree of Doctor of Sciences

Diss. ETH No. 19230

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Koen Langendoen, co-examiner

Examination date: August 27, 2010

Abstract

A Wireless Sensor Network (WSN) is an embedded computation system for distributed sensing of a dispersed phenomenon. It is a distributed system built of autonomous, yet cooperating embedded devices, so-called sensor nodes. Each sensor node provides computational, communication and storage resources and typically operates on limited energy resources. WSNs are often deployed in remote locations for long-term unattended operation. Hence, the validation of correct functioning of the system before the actual installation is of utmost importance.

Validation of WSNs is typically focused on *system testing*, i.e., analyzing system executions including software and sensor node hardware. System testing of WSNs is a complex task. WSNs are distributed systems and have a high degree of concurrency, resulting in a very large state space. The internal state of the sensor nodes is hidden rendering analysis of executions intricate. System testing needs to consider that the operation of a WSN is highly dependent on the environment. As a prominent example, wireless communication depends on environmental conditions, is changing over time and hence unreliable.

This thesis contributes several solutions for testing WSNs. Its goal is to provide automated tool support for executing testcases, extracting meaningful information from test executions, analyzing the monitored information, and checking for the conformance to a specification of expected behavior. To this end, the first part of the thesis focuses on testing functional properties of a WSN. It describes a framework to execute the same test on different test platforms such as simulators and testbeds. Furthermore, an analysis framework is presented that allows a tester to extract behavioral information from test execution logs. A corresponding programming language was developed that simplifies common analysis tasks such as determining the routing paths of a network protocol. The second part of the thesis focuses on testing non-functional properties, in particular power consumption. It presents a test architecture that enables a tester to extract and monitor such properties. Moreover, it describes a formal conformance test for measurements of power consumption and details on its formal foundations. It discusses various optimizations to make the conformance test relevant for practical application and demonstrates its efficiency by a comparison with a state-of-the-art online testing tool.

Zusammenfassung

Ein drahtloses Sensornetzwerk ist ein eingebettetes Rechensystem zur verteilten Überwachung von räumlichen Phänomenen. Es ist ein verteiltes System aus autonomen, jedoch kooperierenden Sensorknoten. Sensorknoten verfügen über Rechen-, Kommunikations- und Speicherressourcen und haben ausserdem üblicherweise ein begrenztes Energiebudget. Drahtlose Sensornetzwerke werden gewöhnlich für einen autonomen Langzeit-Betrieb eingesetzt. Daher ist es wichtig das System vor der Installation auf korrekte Funktionsweise zu überprüfen.

Die Überprüfung von drahtlosen Sensornetzwerken konzentriert sich meist auf Systemtests. Es werden Ausführungen des kompletten Systems, bestehend aus Hardware- und Softwarekomponenten, analysiert. Das Testen von drahtlosen Sensornetzwerken ist eine komplexe Aufgabe: Ein drahtloses Sensornetzwerk ist ein verteiltes, nebenläufiges System und hat daher einen grossen Zustandsraum. Die Analyse ist erschwert, da der interne Zustand des Systems verborgen ist. Systemtests müssen berücksichtigen, dass die Ausführung eines drahtlosen Sensornetzwerks stark von der Umgebung abhängt. Zum Beispiel hängt die Qualität der drahtlosen Kommunikation von den Umgebungsbedingungen ab und ist zeitlich variabel.

Diese Dissertation stellt mehrere Lösungsansätze für Systemtests vor die den Testprozesses zu unterstützen. Dies beinhaltet die Ausführung von Tests, das Aufzeichnen von aussagekräftigen Informationen, die Analyse solcher Testaufzeichnungen und die Überprüfung auf Konformanz mit einer Spezifikation des Verhaltens. Der erste Teil der Dissertation behandelt funktionale Eigenschaften von drahtlosen Sensornetzwerken. Ein Verfahren wird beschrieben das es erlaubt Tests auf verschiedenen Testplatformen, wie einem Simulator oder einem Testbett, auszuführen. Es wird ein Ansatz vorgestellt um Verhaltensinformationen aus Testaufzeichnungen zu extrahieren. Eine daraus entwickelte Programmiersprache erleichtert typische Analyseaufgaben wie das Bestimmen der Routingpfade eines Netzwerkprotokolls. Der zweite Teil der Dissertation beschäftigt sich mit nicht-funktionalen Eigenschaften, im speziellen mit der Leistungsaufnahme. Er präsentiert eine Testarchitektur, die diese Eigenschaften aufzeichnet und überwachen lässt. Ausserdem wird ein formaler Konformanztest der Leistungsaufnahme beschrieben.

Acknowledgement

First and foremost I would like to express my gratitude to Prof. Dr. Lothar Thiele for supporting my thesis and my research. Thank you for the opportunity to come back from industry and for the continuous support and trust during my PhD studies.

I would like to thank Prof. Dr. Koen Langendoen for co-examining my thesis and the time at TU Delft that provided fresh and interesting insights into research and life.

I would also like to thank all my current and former colleagues and friends of the whole TEC group and the ES group in Delft for their company and support. In particular, I would like to thank my office buddies and my running mates. Thanks to all my friends that made my time in Zurich so memorable.

Ultimately, this thesis would have never been possible without the support of my family. Mein Dank geht daher im speziellen an meine Eltern und meine Brüder, die immer für mich da sind.

The work presented here was supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

Contents

Abstract **i**

Zusammenfassung **iii**

Acknowledgement **v**

1 Introduction **1**
1.1 Contributions 2
1.2 Outline 4

2 Wireless Sensor Networks and Testing **5**
2.1 Wireless Sensor Network (WSN)s 5
2.1.1 Sensor node hardware 6
2.1.2 Sensor node software 7
2.2 Data collection application 8
2.2.1 Low-power Medium Access Control (MAC) protocols . 9
2.2.2 Collection Tree Protocol (CTP) 10
2.3 A WSN Model for sensor node placement 11
2.3.1 Related work on sensor node placement 11
2.3.2 Deployment model 12
2.3.3 Evaluation 15
2.3.4 Summary 16
2.4 Validation 17
2.5 Testing 18

3 Testing of Wireless Sensor Network (WSN) applications **21**
3.1 Multi-platform testing 23
3.1.1 Multi-platform test framework 23
3.1.2 Feasibility Study 24
3.1.3 Testcase 25
3.1.4 Discussion and related work 26
3.2 Analyzing test executions 27
3.2.1 Traces, Events and Event Analysis 28
3.2.2 Event Analysis 29

3.2.3 Event analysis operators 29
3.3 Rupeas 32
3.3.1 Domain Specific Language 33
3.3.2 Language implementation 33
3.3.3 Case studies 38
3.3.4 Discussion and related work 41
3.4 Summary 43

4 Test automation for sensor networks 45
4.1 Testing power consumption 46
4.2 Background 47
4.2.1 Error classification 48
4.2.2 Continuous Integration 49
4.3 The Power Testing Architecture 49
4.3.1 Physical parameter extraction 50
4.3.2 Cognitive aids 51
4.4 Formulating tests for power consumption 52
4.4.1 Reference-based evaluation 53
4.4.2 Power unit tests implementation 56
4.4.3 Discussion 58
4.5 Related work 58
4.6 Summary 59

5 Model-based conformance testing of power consumption 61
5.1 Background Theory 63
5.2 Power Trace Testing (PTT) 67
5.2.1 Timed automaton models employed in PTT 67
5.2.2 Reachability check for verifying PTT conformance 71
5.2.3 Compositional modeling of the system model 74
5.2.4 Trace Automaton Optimization 78
5.3 Testing power consumption with TRON 82
5.3.1 Timed input/output conformance relation 82
5.3.2 Trace adapter 84
5.3.3 Sampler process 84
5.3.4 PTT execution with TRON 85
5.4 Case Study 86
5.4.1 Modeling the Harvester 87
5.4.2 Testcases 90
5.5 Empirical evaluation: Results and Benchmarks 92
5.5.1 Power trace models 92
5.5.2 Experimental setup 92
5.5.3 Uppaal results 93
5.5.4 TRON results 95

5.5.5 Uppaal versus TRON comparison 96
5.6 Summary 97
5.6.1 Related Work 98
5.6.2 Discussion 98

6 PTT for communicating sensor nodes 101
6.1 Background Theory 102
6.2 Scalability, an open issue 103
6.2.1 Problem 103
6.2.2 Property of the power trace model: No absorbing loops. 106
6.2.3 Property of low-power (embedded) systems: Recurrent identifiable locations. 109
6.2.4 Segmented power trace testing 111
6.3 Case Study 112
6.3.1 Testing communicating sensor nodes 113
6.3.2 Experimental setup 116
6.3.3 Results 116
6.4 Related work 118
6.5 Summary 119

7 Conclusions 121
7.1 Contributions 121
7.2 Future work 122

Bibliography 124

A Acronyms 139

B List of Publications 141

1

Introduction

A Wireless Sensor Network (WSN) is a novel scientific and industrial monitoring instrument to sense physical phenomena. WSNs are distributed systems built of autonomous, yet cooperating embedded devices, so-called sensor nodes, which each provide computational, communication and storage resources. Advances in micro-electronics and wireless networking have allowed to apply this novel sensing-centric paradigm to a wide variety of applications, e.g., for monitoring flora [LBV06], fauna [MPS+02] and geo-scientific processes [WALJ+06], structures [XRC+04] and for medical applications [LrCC+09]. Deployments of WSNs have shown that the technology is applicable and feasible for very different scenarios. However these deployments have faced various issues in a real-world setting concerning system failures and outages, e.g., due to energy depletion, and unsatisfactory quality-of-service [BRWR10]. While some of the experienced problems are simply due to improper casing and environmental protection, many problems stem from the fact that sensor networks present a new, challenging class of computer systems. WSNs are deployed for unattended and often long-term operation. Hence, validation of correct functioning of the system before the actual deployment is of utmost importance. Validation of WSNs is particularly focused on system testing, i.e., software running on the sensor node hardware. Ideally, each possible state that the system may reach must be explored.

WSNs are distributed systems of sensor nodes. Each sensor node can be in a number of states depending on the complexity of the embedded software and the software's use of different operating modes of the hardware. Since sensor nodes operate autonomously and thus highly concurrent, the resulting number of system states grows exponentially with the number of sensor nodes. Hence, exploring all systems states of a WSN is difficult. A further complication is

that the exact internal state of a sensor node is unknown, since for such an embedded system interaction is only possible through few distinct interfaces. This means that determining the exact state of a WSN is hard. WSNs use wireless technology. Thus, message exchange is stochastic in nature, as wireless communication can fail for various reasons such as interference and multi-path effects. In summary, the huge state space combined with limited visibility into the system and the stochastic nature of communication render validation of sensor nodes complex and challenging.

Since the number of states to be explored is prohibitively large, validation has focused on testing, i.e., a non-exhaustive method. Testing of WSNs tries to find the presence of adverse system states. As such, the software is tested as a whole typically on the actual hardware, yet without the intent to determine the underlying cause of such an adverse system state. Testing has the benefit that actual executions of the real system in a specific environment can be analyzed. This allows tests to explore intricate details such as the effects of the environment, the interaction of hard- and software and the physical properties of the hardware.

The main goal of this thesis is to support the development process with respect to testing. Up to now, research has focused on providing instruments for the evaluation in form of dedicated testbeds [DBT+07, WASW05, HKWW06, EAR+06, HHP+08], simulators [LLWC03, ODE+06, Bou07], and emulators [TLP05]. This thesis extends this work by focusing on the automation of the testing process given test platforms and by providing novel methods and tools for the evaluation of test executions in order to determine whether they show valid behavior with respect to a specification. The evaluation of executions may concern functional properties, such as the amount of successful collected data from the WSN, and non-functional properties, such as the energy-efficiency of the system.

1.1 Contributions

This thesis presents methods and tools for testing WSNs and thereby supporting the development process of WSN software. As such the contributions of this thesis in order of appearance are listed below.

1. **Multi-platform testing**
   When developing a sensor network application, various testing tools are employed at different stages of development: In the beginning an abstract model is simulated. During development, software is run on real sensor nodes on a testbed in different test configurations. A *test framework* allows the same test to be executed on different test platforms. Testcases need to be devised only once and the evaluation can be formulated and performed

independently. A test framework is presented using two state-of-art tools: a simulator and a testbed.

2. **Event analysis for the analysis of test executions**
   Given a platform-independent test framework, individual test executions necessitate an analysis of individual runs. *Event analysis* provides a platform- and application-independent analysis approach based on the notion that during a test all sensor nodes log events that comprise information about the execution. A new programming language, called Rupeas, is presented for analyzing test executions that exploits the event analysis approach. Based on different testcases, it is shown how event analysis can be used to evaluate test executions of a WSN application in both, testbed executions and simulations.

3. **Architecture for automatically testing for power consumption**
   Automation is one of the key requirements for software testing. However, typical test automation does not consider non-functional properties such as power consumption. Hence, an architecture needs to be devised that automatically executes tests, measures power consumption of the system and checks whether the measurements conforms to a specification. To this end, an approach for detecting power consumption measurements deviating from a reference-based specification is presented.

4. **A model-based approach for testing power consumption**
   A ramification of a reference-based approach for checking power consumption is that it requires the specification of a single, deterministic reference for each testcase. In contrast, a novel, model-based approach is presented that presents a general approach, which can be used for testing of different physical phenomena, e.g., for testing WSNs. The method is based on modeling a physical quantity and the system as timed automata extended with data variables. The theoretical background of the methodology is presented along with implementations for investigating the power consumption of a sensor node.

5. **Testing communicating sensor nodes**
   When considering a model-based approach for testing sensor networks, an integral part is the communication models of the individual sensor nodes. The modeling of the communication has a significant effect on the tests as well as on analyzing the results of a test execution. Based on a sample application, models, issues and a mixed-testing approach are presented in order to allow for model-based testing for power consumption of WSNs.

The work presented in this thesis is based on the following book chapters, journal articles, conference papers ad technical reports: [BDL+07, BLM+09,

LWMB09, WBHB08, WBLT08, WBH07, WBT08, WBYT08, WLT09, WPBT07, WPL+08, WPT09, WPT10]

1.2 Outline

The outline of this thesis is as follows: In the following chapter, details on wireless sensor network state-of-the-art and background on software testing is provided. Chapter 3 presents multi-platform testing to facilitate testing on different platforms and introduces a novel methodology for analysis of log files based on an event abstraction. The event abstraction is implemented in a novel programming language and applied to testing a WSN application. Chapter 4 describes a novel architecture for automatically testing sensor networks, in particular their power consumption. Apart from the test architecture, a novel method is presented to evaluate measured power consumption of a WSN test execution. This concept is further elaborated on in Chapter 5, where a model-based conformance test is presented that allows a system to be tested for power consumption. The theoretical background is presented as well as implementations using two timed verification tools. Results on testing power consumption of a sensor node are presented and compared for the two tools. Finally, Chapter 6 extends these concepts to communicating sensor nodes. It details further optimizations based on domain-specific properties. Chapter 7 concludes the thesis and provides an outlook on future work.

2

Wireless Sensor Networks and Testing

This chapter presents a background on sensor networks and testing. In particular, it describes the hardware and software used in sensor networks and presents an example of a sensor network application, which is used as a test application in many of the following chapters. Additionally, this chapter details on software testing as a specific method for performing validation.

2.1 Wireless Sensor Network (WSN)s

WSNs are embedded sensing systems deeply integrated into the environment to cooperatively monitor a dispersed phenomenon. Data is collected and forwarded to one or more sink nodes which typically have additional processing, energy and storage resources and provide the ability to connect to a secondary network acting as gateways. WSNs are used in various application areas [RM04] with different requirements and environmental characteristics.

WSNs are networks built of (typically homogeneous) sensor nodes. In principal, each sensor node provides: (i) sensors that collect some data and ADCs to convert analog measurements into the digital domain, (ii) a timer subsystem for time-driven functionality, (iii) a small microprocessor with limited memory, (iv) communication capabilities using RF technology, (v) limited energy resources typically provided by a battery and (vi) flash storage. The sensor node software adds a minimal operating system including hardware drivers

	Tmote	IRIS	TinyNode 184
MCU	MSP430F1611	ATmega1281	MSP430F2417
Architecture	16 bit	8 bit	16 bit
Clock (max)	8 MHz	16 MHz	8 MHz
Program Flash	40 kB	128 kB	92 kB
RAM	10 kB	8 kB	8 kB
Radio	TI CC2420	AT86RF230	SX1211
802.15.4	Yes		No
Frequency	2.4 GHz		868/915 MHz
Data Rate	250 kbps		< 200 kbps
Serial Flash	1024 kB	512 kB	

Table 2.1: Different sensor node platforms and their components. The MSP430 family is manufactured by Texas Instruments, the Atmel Family is manufactured by Atmel. The AT86RF230 radio is manufactured by Atmel, the SX1211 is manufactured by Semtech. All memory sizes are shown in byte (B) or kilobyte (kB).

and a protocol stack typically up to the networking layer, and an application layer.

2.1.1 Sensor node hardware

There have been various different WSN hardware platforms used in research, but most of these sensor nodes fall into the *mote*-class [HSW+00]. These are built of Commodity-Off-The-Shelf components integrated onto a board with a small form factor. The cost of application-specific hardware renders custom design for individual projects economically infeasible. Typical sensor nodes feature a Microcontroller Unit (MCU) with a very low sleep power e.g., the Atmel AVR or the TI MSP430 as shown in Table 2.1. The processing power of the 8 or 16 bit microcontrollers is fairly limited, comparable to microprocessors in the 1970s.

Sensor nodes feature low-power radios for wireless communication in the ISM bands[1]. In particular, radios based on the IEEE 802.15.4 standard have been popular. Radios provide abundant bandwidth for typical sensing applications with a few bytes every couple of minutes [BGH+09] as indicated in Table 2.1. However, this bandwidth comes at a cost, since the radio is usually the major consumer of energy. Table 2.2 indicates that the radio consumes an order of magnitude more power than the MCU and multiple orders of magnitude more power when communicating than when being in a low-power mode. Additionally, the radio often consumes a large amount of power regardless of

[1] The ISM radio bands are internationally reserved for the unlicensed use of RF technology for industrial, scientific and medical purposes.

Components		Platform current draw (mA)			
MCU	Radio	Tmote	MicaZ	IRIS	TinyNode
on	RX/Idle	23.0	27.7	24.0	3.5
on	TX	21.0	25.4	25.0	25.1
on	low-power	2.4	8.0	8.0	4.2 (@8 MHz)
low-power	low-power	$21.0 \cdot 10^{-3}$	$16.0 \cdot 10^{-3}$	$8.0 \cdot 10^{-3}$	$2.5 \cdot 10^{-3}$

Table 2.2: Current draw values for different states of the Microcontroller (MC) and the radio of a Tmote Sky, a Crossbow MicaZ, a Crossbow Iris node and a TinyNode 184. Datasheet values at highest transmission power are shown for the radio. These values are measured for a constant supply, i.e., they are proportional to the power consumption of the sensor node.

whether it receives, sends or just listens to the channel as exemplified with the CC2420 radio. For this reason, radios are often transferred into a low-power mode, where no communication is possible. This so-called duty cycling trades off available bandwidth for reduced energy consumption. Low-power radios provide limited transmission range in the order of tens of meters (without special antennas). Hence, to cover a large area it is not possible that every node can directly communicate with a sink node. Rather a multi-hop, mesh-network needs to be established. The mesh allows collaborative forwarding of the data to the sink.

Sensor nodes are typically energy-constrained, i.e., running on batteries. In some cases, energy can be harvested from the environment. However, except for specific scenarios [CVS+07], harvested energy is not abundant and hence there is still a tight bound on available energy.

Memory on sensor nodes is limited. In addition to volatile Random Access Memory (RAM) memory, sensor nodes provide additional non-volatile flash storage. This additional data memory allows for temporarily storing measurements, e.g., when communication is temporarily not possible or for archiving data for post-deployment validation [BGH+09].

2.1.2 Sensor node software

The availability of standard platforms and economical restrictions of custom hardware design has resulted in an increased focus towards the embedded software. Sensor node software can be categorized in two main classes: (i) the communication stack including application logic and (ii) the operating system including drivers. In the following, an OSI network model for the protocol stack is used. Hence, the physical layer is provided by the radio hardware, the data link layer is handled by the MAC protocol and the network layer uses typically a many-to-one (convergecast) routing protocol. From layers 4-6, only the Transport Layer has seen limited use for applications where data needs to

be transferred reliably [PG07]. The application layer comprises the application logic responsible for local sensing, processing of data and passing data to the network layer.

Software needs to exploit low-power states of the hardware components to drastically minimize the power consumption. Application-specific optimizations are pushed towards the software design, in particular the communication protocols, rendering protocol design a lively field of research. As described, the radio is typically the major consumer of power on a sensor node. MAC Protocols are responsible for duty-cycling the radio. Hence, energy-efficient MAC protocols have been extensively researched [Lan08]. An additional consideration for the protocol stack is reliability. As wireless communication is influenced by communication losses due to (internal and external) interference, multi-path fading, etc., reliability is added to the protocol stack. Mechanisms include (single-hop) message acknowledgments combined with retransmissions from the MAC or Routing Protocol or even end-to-end acknowledgments provided by a transport layer [PG07].

While the individual protocol layers are conceptually independent, there is very close interaction and dependencies between individual protocol layers of a sensor node [ZFWT10]. Optimization of the stack often requires a comprehensive analysis by studying cross-layer dependencies, e.g., [MLM+05].

Apart from the communication stack, the second class of sensor network software is the operating system, which typically includes drivers for given sensor node platforms. Operating systems for WSNs are designed very lean due to the scarcity of resources. Hence, these operating systems, e.g., [LGH+05, DGV04], restrict themselves to minimal capabilities such as interrupt handling, thread scheduling and simple computation such as packet processing. Since the underlying hardware architecture does not provide sophisticated mechanisms, such as address translation and access protection, many features that typical operating system offer cannot be provided on a sensor node.

The dominantly used, open-source TinyOS [LGH+05] is a prominent example of a sensor network operating system. TinyOS natively provides a two-level concurrency model with interrupt routines, which can be either preemptive or non-preemptive, and a separate non-preemptive queue for deferred procedure calls, so-called tasks. A separate preemptive threading library [KLP+09] additionally allows programmers to use thread-based programming. Additionally, TinyOS provides a set of libraries, including various MAC, routing and security protocols as well as different applications.

2.2 Data collection application

In order to evaluate many of the novel methods and tools presented in this thesis, a test application is necessary, which is representative for a whole class

of applications. Hence, data collection is considered, which is the principal task for sensor networks in monitoring, e.g., of the environment [MPS+02, BGH+09] or buildings. A test application from the predominant operating system TinyOS is selected: This application, the so-called *MultihopOscilloscope* application, performs local measurements (like an oscilloscope) and sends aggregated measurements back to a sink via multiple hops. The sink forwards all received packets over the serial port to a host PC or a gateway. Data collection is evaluated based on the *data yield*, i.e., how much of the data sensed and sent on each of the sensor nodes actually arrives at the sink node.

MultihopOscilloscope uses the Collection Tree Protocol (CTP) [GFJ+09] on the network layer, further described in Sec. 2.2.2. The basic variant of MultihopOscilloscope used in this thesis uses a CSMA protocol on the data link layer. In a nutshell, a CSMA protocol listens to the carrier before sending a packet. However, collisions may still occur due to hidden-terminal effects. A CSMA protocol performs no duty-cycling.

In some testcases an energy-efficient variant of the MultihopOscilloscope application, the so-called Harvester [LWMB09], is used as a test application. The energy-efficiency of Harvester comes from the use of a low-power MAC protocol described below.

2.2.1 Low-power MAC protocols

Energy-efficient operation of the MAC layer necessitates trading off bandwidth of the radio for energy consumption by periodically turning off the radio. This so-called duty-cycling determines the energy consumption as well as the available bandwidth. However, two nodes that want to communicate still need to perform a rendezvous, i.e., one sending and the other one being in receive mode at the same moment in time.

There are basically two fundamental types of MAC protocols: Those that organize the rendezvous times and those that do not. The latter is the class of the so-called random-access MAC protocols for which nodes independently select a point in time, when they wake up for receiving a packet. An energy-efficient subclass of random-access protocols is the family of Low-Power Listening (LPL) protocols. In these protocols, the receiver periodically, yet independently of its neighbors, wakes up every T_W (the wake-up time) and listens for ongoing traffic. A sender ready for a transmission starts to send a preamble at a given point in time to signify intent of transmission to any waking up sensor node in the environment. All nodes sampling the preamble stay on until the actual packet transmission. In order to guarantee that the sender can address its receiver, the preamble must last for (a little longer than) T_W. After the preamble is finished, the actual data packet is sent [PHC04].

The LPL scheme can be further improved when using a packet-based radio as in XMAC [BYAH06]. The preamble consists of individual packets that

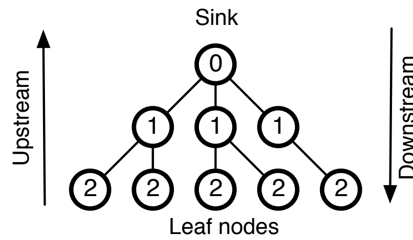

Figure 2.1: Simple Tree built on the distance vector. The distance metric is the hop count.

contain the address of the intended receiver. Hence, any node that is not addressed can go back to sleep. Additionally, packets can be directly acknowledged allowing the receiver node to give an early acknowledgment as soon as it receives the first preamble packet. This reduces the average preamble length by a factor of two. A further step to increase energy efficiency is to synchronize the sender to its receivers. WiseMac [EHD04] uses packets extended with time information to estimate the wake-up times of communicating nodes. In turn, this estimation can be used to start transmitting packets only shortly before the intended receiver wakes up.

Note that all of these optimizations only work in unicast transmissions, i.e., when there is a single addressed receiver. For broadcast operations, all LPL protocols need to transmit the message for a whole wake-up time T_W. Hence, for energy-efficient MAC protocols, broadcast messages are considerably more expensive than unicasts [Mei09]. Harvester uses a synchronized LPL MAC protocol similar to WiseMac [EHD04] on packet-based radios, sending the data packet directly instead of a dedicated preamble packet [LWMB09].

2.2.2 Collection Tree Protocol (CTP)

In the test application, the Collection Tree Protocol (CTP), which is included in the TinyOS distribution, is used for data collection. Data collection relies on the network layer establishing a routing tree that allows nodes to send their data to one or more sink nodes in a multi-hop fashion.

While CTP features elaborate mechanism for efficiently maintaining a routing tree even over unreliable links, for this discussion it is only important that it is a distance-vector based protocol, i.e., each node maintains a notion of distance to the destination, the sink. Nodes periodically broadcast their distance to the sink via so-called beacons, so that neighbors can send their data to a node that is closer to the sink (or the sink itself). An example of such a tree is depicted in Fig. 2.1. In this case, the distance metric is the hop count. CTP has a more fine-grained distance metric that also evaluates the link quality to a neighboring node to determine the distance.

In conclusion, the MultihopOscilloscope application refers to collection using CTP and a Carrier Sense Multiple Access (CSMA) MAC protocol. Similarly, the Harvester application performs data collection using CTP on top of a synchronized LPL MAC protocol.

2.3 A WSN Model for sensor node placement

For analyzing WSN deployments, models can support the design process. In the following, the deployment problem of sensor nodes is described, which can be stated as: "How many wireless sensor nodes should be used and where should they be placed in order to form an optimal wireless sensor network (WSN) deployment?". The problem of how to distribute sensor nodes to cover a certain area with as few nodes as possible but still provide reliable communication paths from each node to a data sink is studied. Before presenting a model based on state-of-the-art research on wireless communication and how it can be utilized with randomized search heuristics, related work is discussed.

2.3.1 Related work on sensor node placement

Several approaches for the deployment of WSNs have been proposed in the literature. However, there is no work employing a realistic deployment model for nodes and the environment and at the same time exploring the intricate trade-offs between connectivity and cost, while guaranteeing coverage of the deployment area.

For example, [DCI02] and [SY05] present algorithms to improve the deployment coverage. Both papers do not consider deployment connectivity and the corresponding trade-offs. [WXZ+03] present the integration of communication and sensing coverage whereas [Jou06] looks at coverage and lifetime. Both works use communication models that are limited to a simplistic homogeneous Euclidean distance model. [BKX+06] proves the asymptotic optimality of a stripe-based deployment pattern for different ratios of sensing range to communication range. The latter approaches of [WXZ+03], [Jou06], and [BKX+06] are based on simplifying assumptions, as discussed by [KNG+04]. [RVMM05] uses a more realistic communication model and, in addition, investigates the trade-offs with respect to energy consumption. However, only points on a spatial grid are considered as possible node positions.

None of the related work addressing the coverage problem considers the complex trade-off between reliability of communication and deployment costs. To the best of our knowledge, only [KGGK06] consider coverage, cost and communication in a realistic scenario. The authors present a polynomial-time, data-driven algorithm using non-parametric probabilistic models called Gaussian Processes. Since their work requires sensor and link quality data

collected at an initial deployment, the work of [KGGK06] complements the approach presented in this thesis, as the presented approach can determine an optimized deployment without any preceding data collection.

2.3.2 Deployment model

The considered deployment model is divided into two parts, an environment model and a model for a set of homogeneous sensor nodes. The environment is represented by a data sink to which all the sensor readings need to be communicated and an area of interest that is to be monitored. This area of interest is outlined by a polygon and represented by a set of points of interest. The area of interest is covered by sensors if every point of interest lies within the sensing range of at least one node. Note, that the proposed formulation explicitly allows sensor nodes outside the region of interest, although they only contribute to the improvement of routing paths.

Communication: The radio model is derived from the prior work of Zuniga et al. [Zun04, ZK07] and the work of Zhou et al. [ZHKS06]. Zuniga et al. present in [ZK07] an analysis of packet reception rates in low-power wireless links. In particular, the authors present an analysis of asymmetry in wireless links. Additionally models for the different regions in wireless communication are presented: the connected region, where connectivity is almost perfect, the transitional or gray region, where reception is very dynamic and the disconnected region where communication is not possible. While the expected packet reception rate decreases with distance, a significant variance in the transitional region requires a stochastic perception by defining probability thresholds for low/high probability of low/high packet reception rates [2]. The transitional region coefficient defines the ratio of the transitional to the connected region. The transitional region coefficient is independent of noise floor and output power. These models were previously used in other WSN communication studies such as [OSFC07]. [ZHKS06] presents the Radio Irregularity Model (RIM) to account for radio irregularities in WSNs. This is mainly due to anisotropic path losses caused by the non-uniformity of the environment, and heterogeneous sending powers, mainly due to device differences caused by manufacturing variations. They present the effect of this model on protocol layers, such as MAC, routing, localization and topology control. They introduce three different parameters: The degree of irregularity (DOI) models the anisotropy by describing the maximum path loss percentage variation per unit degree change in the direction of propagation and the corresponding variation with incremental changes. The Variance of Sending Power (VSP) and the Variance of DOI values (VDOI) account for the heterogeneity of nodes.

[2] The disconnected region typically has a packet reception rate threshold (PRR) larger than 0, since links with low PRR incur too many communication losses to be of any practical use.

Based on this priori work, the deployment model for each node $k \in \{1, \ldots, n\}$ includes:

- A node's position as 2-dimensional coordinates (x_k, y_k),
- a radio model to describe transitional regions in communication. The packet reception rate is computed in detail as a function of the distance d between the nodes as follows, cf. [Zun04]. The signal to noise ratio (SNR) is defined as

$$\gamma_{dB}(d) := P_t - PL(d_0) - 10\eta \log_{10}\left(\frac{d}{d_0}\right) + \mathcal{N}(0, \sigma) - P_n$$

The packet reception rate (PRR) follows as:

$$PRR(d) := (1 - \frac{1}{2} e^{-\frac{\gamma(d)}{2} \cdot \frac{1}{0.64}})^{8f},$$

where $P_t, PL(d_0), d_0, \sigma, P_n, f$ are constants. Note that $\gamma(d)$ is not used in dB in the equation and must be calculated as $10^{(\gamma_{dB}/10)}$. η is dependent on the degree of irregularity (see below), which is a function of the angle. It follows that:

$$\eta = \eta(DOI) = \eta_0 \cdot K_i,$$

where K_i is the path loss coefficient in the direction of the transmission,
- a degree of irregularity (DOI), describing the anisotropy of radio communication due to the anisotropic medium and hardware variances, which is used to adjust the path loss η. DOI is defined as the maximum path loss percentage variation per unit degree change in the direction of the radio propagation. In [ZHKS06], K_i ($i \in \mathbb{N}$) is defined as a coefficient to represent the difference in path loss in different directions [3]:

$$K_i := \begin{cases} 1, & \text{if } i = 0 \\ K_{i-1} \pm R * DOI, & \text{if } 0 < i < 360, \\ & \text{where} \\ & |K_0 - K_{359}| \leq DOI \end{cases}$$

R is a random variable, uniformly drawn from $[-1, 1]$. The code shown in Listing 2.1 is used in the presented node model to determine DOI.
- an elliptic sensing region per sensor type, defined by the sensing radius r_{sense},
- an area of interest, in this study outlined by a polygon and represented by a set of points of interest.

The settings for individual parameters are described in [WBH07].

[3] Although not mentioned in the original paper, there needs to be the requirement, that $K_i \geq 0$. This is not reflected in the pseudo-code of the algorithm below.

```
K[0:359] = (1,1,1,1,...)
  for i in 0 to 359 do:
   random = uniform(-1,1)
   K[(i+360-2) mod 360]+= 0.4*random*DOI
   K[(i+360-1) mod 360]+= 0.8*random*DOI
   K[(i+360  ) mod 360]+= 1.0*random*DOI
   K[(i+360+1) mod 360]+= 0.8*random*DOI
   K[(i+360+2) mod 360]+= 0.4*random*DOI
```

Listing 2.1: Code for determining DOI. `uniform` is a function that returns a random number from a uniform distribution in the interval defined by its parameters.

Furthermore, there are two optimization criteria for the deployment problem additional to the requirement that the deployment area must be covered by sensor nodes.

Sensor Cost: Each sensor node that has to be placed causes costs, i.e., for production, deployment, and maintenance. Since one is interested in a cost-effective solution, the first optimization criterion is to minimize these costs and thereby the number of nodes. In a first approach, a cost of '1' is associated with each node. Therefore, the number of nodes n is used as the first optimization criterion:

$$f_1 = n \tag{2.1}$$

Transmission Failure Probability: The sensor readings need to be continuously communicated from the nodes to the data sink. Thus, each of the nodes needs a reliable routing path to the data sink; if the sink lies outside of the radio range of a specific node, its routing path contains intermediate nodes which forward the message to the sink. Since wireless communication is susceptible to communication failures between nodes, e.g., due to interferences or node failures, not only the reliabilities of the best routing paths are necessary to be optimized but redundant transmission paths of high reliability as well. Instead of maximizing the connection reliability, here the dual criterion of minimizing the transmission failure probability is considered:

$$\begin{aligned} f_2 &= \frac{1}{W} \cdot \sum_{j=1}^{N_{red}} w_j \cdot (1 - p_{worst,j}) \qquad (2.2) \\ \text{with } W &= \sum_{j=1}^{N_{red}} w_j \end{aligned}$$

Equation 2.2 scores the difference between the worst transmission path $p_{worst,j}$ on redundancy level j to an optimal path with transmission probability 1. Therefore, minimizing this criterion ensures that there is a preference for node placements resulting in high transmission reliabilities; it explicitly allows for assigning different weights w_j to connections on different redundancy levels j. In turn, f_2 is normalized with the sum of these weights W.

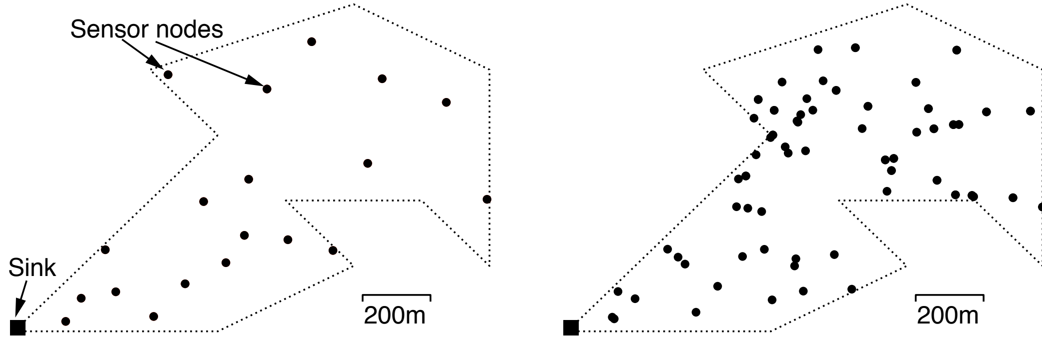

Figure 2.2: Two non-dominated solutions of sensor node placements for the same area of interest outlined by a polygon. Nodes are marked by circles and the sink is indicated by a square. Both solutions are part of the set of non-dominated solutions shown in Fig. 2.3. The left solution corresponds to an objective vector of $(f1, f2) = (18, 0.5827)$ while the network on the right has objective values $(f1, f2) = (61, 0.0016)$.

The path reliabilities of the N_{red} most reliable paths between all nodes i and the sink are computed as follows: For each node i, the most reliable path to the sink and store its corresponding reliability $p_{i,1}$, are determined by using Dijkstra's algorithm. Afterwards, all nodes of this path except source and sink are deleted; this procedure iteratively repeats until N_{red} paths are found or no longer a path exists (if less than N_{red} paths are found, all missing paths are assigned a probability of zero).

2.3.3 Evaluation

The presented model can be used for exploring the design space of WSN deployments with respect to node placement. Trade-offs between the number nodes and the connection reliability can be explored by using an off-the-shelf Multiobjective Evolutionary Algorithm (MOEA) [Deb01]. In this work, the MOEA IBEA by [ZK04] is used, as it is provided in the PISA framework of [BLTZ03]. Some domain-specific adaptations of the MOEA are performed. For details the interested reader is referred to [WBHB08], which elaborates on the adaption of the MOEA to the new search space, including a novel variation operator based on Voronoi-diagrams.[4]

In optimization problems with many objectives, solutions are compared using the pareto dominance relation [Deb01]. A solution dominates another solution if it is not worse in any of its objectives and better in at least one of them. The MOEA was evaluated based on a test scenario, i.e., a deployment of sensor nodes in an area of interest as shown in Fig. 2.2. Fig. 2.3 shows that the MOEA

[4]The original work contains an extensive parameter evaluation. In the context of this work, only the best parameterization with crossover probability $\kappa = 1.0$ and mutation ratio $\rho = 3 : 1$ is discussed.

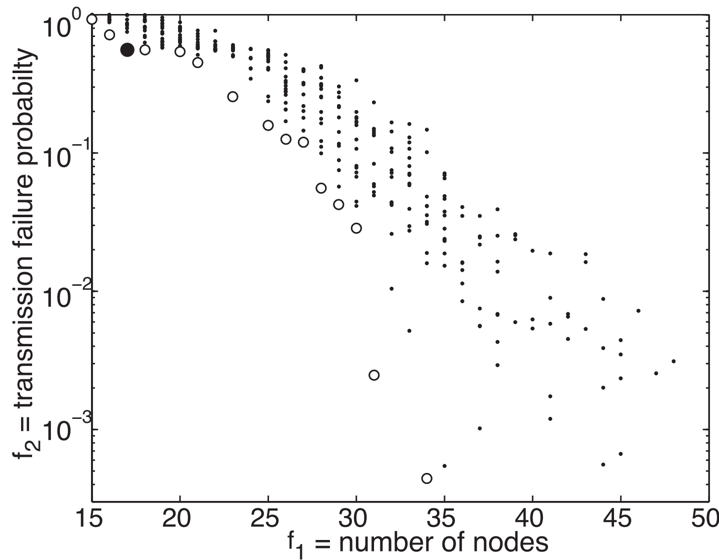

Figure 2.3: Non-dominated solutions. To improve readability, the y-axis is plotted in log-scale. Solutions that are non-dominated over all runs are depicted as large circles. As an example, the dark circle depicts the solution shown on the left of Fig. 2.2.

provides a set of non-dominated solutions for this scenario. Figure 2.2 depicts two example solutions from this non-dominated set exemplifying the trade-off between few nodes and a high connectivity. As an example, the right solution shows that due to the requirement of redundant paths to the sink, multiple nodes may be placed side-by-side in spaces that are critical for routing. The left solution corresponds to an objective vector of $(f_1, f_2) = (18, 0.5827)$ while the network on the right has objective values $(f_1, f_2) = (61, 0.0016)$.[5]

2.3.4 Summary

The proposed MOEA shows its capabilities in identifying a broad range of trade-offs between number of nodes and transmission reliability (see Fig. 2.2 and Fig. 2.3) while satisfying the constraint of sensor coverage. The presented test scenario indicates that the MOEA provides valuable support for planning a sensor network deployment. A specific real-world application where this support would be valuable is the design of a WSN for fire detection: for this type of WSN it is required to guarantee reliable data transmission on at least two redundant paths to ensure that no fire alarm gets lost. On the other hand, the number of nodes affects recurring costs: maintenance is expensive, especially when considering that some of the nodes may be placed in locations that are difficult to access. This is where the proposed MOEA can help by providing

[5] In Fig. 2.3, the left solution is indicated by a dark circle, the right solution is not depicted since it is outside the plotted area.

good trade-off solutions that help the sensor network planner to decide on the most desirable solution.

The planning of a deployment is merely the first step. The major challenge for a sensor network is a correct, energy-efficient and autonomous operation. Hence, the main question addressed in this thesis is how it can be checked whether a deployment of a WSN system actually works.

2.4 Validation

The process of checking whether a WSN system deployed in a given environment operates correctly is called *validation*. More specifically:

Definition 2.1 (Validation of Wireless Sensor Networks)**.** *Validation concerns the process of checking whether a deployment of a WSN satisfies its specification. The specification includes different functional and non-functional properties that need to be ensured under differing environmental conditions.*

Various classes of methods have been previously proposed to address validation of generic computer systems. Each of these classes has a set of idiosyncrasies that make them more or less suitable for use in WSN validation. The major classes include:

- **Formal verification:** Formal verification uses mathematical descriptions of the system $\mathcal{M}$ as well as desired properties ϕ and verifies that the system satisfies these properties: $\mathcal{M} \models \phi$. As such, formal verification is an exhaustive method. The two main approaches for formal verification are (a) enumeration and search of the state space of a model or (b) deduction on inference systems, i.e., using axioms and inference rules [KS08]. This class of methods uses abstraction to trade off complexity, computability and fidelity of models.
- **Static analysis:** Static analysis abstracts away from actual executions of a system and statically approximates a set of possible executions. The approximation renders such methods inexact. Due to the approximation, some properties may be true for a system, but not in its approximation or vice-versa. Analysis trades off complexity, computability and precision.
- **Testing:** Testing uses executions of the system and checks specific properties based on information about the executions. Hence, testing typically requires some instrumentation to extract information. Instrumentation may however interfere with the execution. This interference is particularly likely for access-limited embedded systems. However, tests on real devices can reveal problems created by the intricacies of the interaction of hard- and software that have been the root cause for failed deployments [BRWR10, BISV08, CLWL06, LBV06]. **Measurements** can

be integrated with tests to quantify physical properties of a system execution. Measurements can often be performed without instrumentation, but require additional, costly equipment.

- **Simulation:** Simulation uses models of a system and its environment on different abstraction levels. Models are executed to either study specific runs and their properties, e.g., for testing, or may be used for statistical evaluations. Simulation requires executable models necessitating assumptions about a system and its properties. Simulations can trade-off complexity, i.e., simulation speed, for fidelity. For both, simulations and system executions, only a subset of possible states can be checked in practice; hence, both are non-exhaustive.

For validating a WSN it is of utmost importance to validate every aspect of the system in particular implementation details such as hardware/software interaction. Additionally, WSNs are inherently best-effort systems due to their use of the wireless medium and typically accept violation of properties with a low probability, e.g., there is no guarantee on successful message transmissions. Hence, the focus of this thesis is on validating *different aspects of the realization* of the system such as physical properties of sensor nodes. Rather than being exhaustive, deficiencies due to intricacies of the implementation shall be uncovered. For detecting deficiencies in design, implementation and of physical properties of the system, testing is the primary choice. While this thesis focuses on testing, different validation methods are integrated to assist sensor network development. As such, Chapter 3 presents an integration of simulation and testing in a framework, while Chapter 5 shows the integration of testing, taking measurements and formal methods to test the power consumption of sensor nodes.

2.5 Testing

Testing is a very general term mostly referring to an execution of the software and the evaluation of a test run given a test criterion. Myers [Mye79] defines testing as *the process of executing a program with the intent of finding errors*. To better understand the exact semantics of the definition, an *error* is defined as the system entering a state that deviates from the expectations with respect to some criterion. This erroneous state is triggered by an underlying *defect* in the software; however not all defects generate an error [Par97]. In order to determine erroneous behavior a specification of expected outcome is necessary. Such a specification may be employing formal models (cf. Chapter 5) or merely the assumptions of the developer. In this thesis, the definition of testing is refined to:

Definition 2.2 (Testing of Wireless Sensor Networks)**.** *Testing is the process of executing a program with the intent of finding errors of the WSN system comprising hardware and software with respect to a specification of observable behavior.*

Basically, testing is experimentation using dedicated testcases. A testcase is an execution in a specific environment and under given operating conditions, e.g., typical or corner-case scenarios. In practice, testing can never be exhaustive and cannot ensure complete correctness. Rather, specific testcases are selected in order to increase the confidence in the quality of the software.[6] Testing is the dominantly used validation method and is one of the most time-consuming tasks in software development typically estimated as taking from 30% up to 70% of the development process [Bei90, PY08, Tre08].

Testing of sensor networks can be perceived as *black-box* testing. Embedded systems, e.g., sensor nodes, only interact via dedicated inputs and outputs. Low-level interfaces such as JTAG, UART or I^2C are used for access to internal state. Software can be instrumented with test monitors to expose some internal state [BRWR10]. This necessitates off-system logging, since RAM is considerably limited on the sensor nodes and flash access can interfere with system execution. Similarly, on sensor node platforms sharing a bus for the serial interface and the radio such as the Tmote Sky, monitor output may also interfere with the communication stack. In order to avoid system perturbation, test monitors are used restrictively to the smallest set of outputs required to deduce test success. Hence, access to an embedded system must be limited necessitating black-box testing. This also implies that monitoring, test outputs and thus test evaluation are highly application-specific.

Testing a particular run of a black-box system relies on input-output oracles to evaluate whether a given execution was erroneous or not. Such oracles or checkers can be on functional properties, which is addressed in Chapter 3, or non-functional properties, as described for power consumption starting from Chapters 4.[7]

The evaluation of test runs, addressed in the following chapters, is a hard problem: Rice's Theorem explains that in general, any nontrivial property about software running on a computing system is undecidable.[8] The state-space explosion and the black-box nature of WSNs render exploring all systems states as well as determining the exact state of a WSN difficult. Additionally, in WSN testing non-functional properties such as timing constraints or power consumption have to be considered. Moreover, the stochastic nature of wireless communication exacerbates the complexity of testing.

[6]In theory, there are test generation algorithms that are sound and exhaustive [LMN04, ST08]. Nevertheless, only a subset of tests will be selected for actual execution.

[7]In the software testing literature, non-functional properties are often referred to as performance tests.

[8]More rigorously, any nontrivial property about the language recognized by a Turing machine is undecidable due to the halting problem.

3

Testing of Wireless Sensor Network (WSN) applications

When developing a WSN application, there are several possible options in the development process. The typical goal is an application running autonomously on tens to hundreds of WSN nodes. Usually, such a large-scale application is developed by simulating a small number of nodes first. After the simulation passes a set of tests, the developer faces two options: either the application is refined and implemented on a testbed, or the simulation is extended to include more WSN nodes. Detailed simulation and emulation of WSNs requires significant computational resources. Hence, large-scale WSNs can be simulated only with a reduced accuracy and fidelity of the simulation results with reasonable effort. In general, porting an application that has been validated in simulation to the testbed is not trivial, due to inaccurate simulation assumptions, limited debugging capabilities of the testbed, and the resource constraints on the wireless sensor nodes. In summary, various testing tools, so-called test platforms, are employed at different stages of development as illustrated above with simulation and testbed experiments. However, tests are developed individually for each of the different test platforms.

Motivated by many reported pitfalls [CLWL06] and discussions with industrial partners, it is argued that a systematic development approach accompanied by an end-to-end test methodology is key to build sustainable WSN systems. The proposed new test methodology enables a development team to continuously monitor the correctness of an implementation. The implementation of the new methodology in a *test framework* allows for testing WSNs on different platforms as shown in Fig. 3.1 by exploiting so-called test plat-

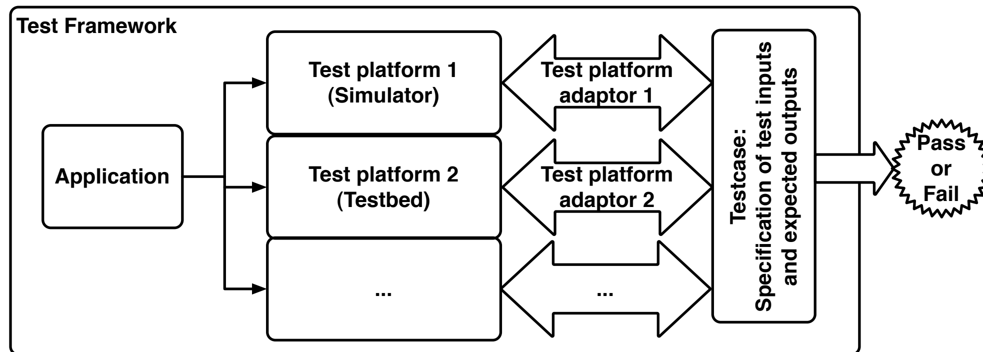

Figure 3.1: The test framework allows testcases to be executed on different platforms by exploiting test platform adaptors. A unified specification of test inputs and expected outputs allows for checking executions on different platforms such as a simulator and a testbed.

form adaptors. The unified test specification, which includes test inputs and expected outputs, are bundled with the WSN software into a testcase. This methodology enables automated testing.

In the first part of this chapter, the methodology and its implementation in a framework for testing in a simulator and on a testbed is presented. A crucial part of the test framework (and testing in general) is the analysis and validation of the information logged during a system execution. Test platforms, simulators such as TOSSIM [LLWC03] and testbeds such as Motelab [WASW05] and the Deployment Support Network (DSN) [DBT+07], feature different logging mechanisms and formats. In all cases, the test data collected by the test monitors during a test execution are centrally available for offline analysis. However, system tests result in a large amount of log data, necessitating adequate tools to analyze the logs. An analysis needs to consider the details of the application, which data is logged, and the test platform. However, the analysis of logged data is typically performed with ad-hoc scripting, which lacks rigor and hence reusability.

To this end, the second part of this chapter presents a new programming language for analyzing log data of WSNs. The *Rupeas* language, a **Ru**by **P**owered **E**vent **A**nalysi**s** facilitates analysis of WSN logs by using an *event abstraction* for each log message. In turn, the log or *trace* is a set of such events. Rupeas provides operators to process sets of events to extract behavioral information: For example, starting from individual send and receive events logged while routing packets, a Rupeas query can extract the actual routing paths of each packet and determine whether and where a packet was lost along the way. By exploiting a domain-specific abstraction, Rupeas provides a simple and concise notation. Rupeas allows users to analyze executions from different platforms, i.e., it is agnostic of actual test platforms and logging mechanisms and analyses generic log files.

3.1 Multi-platform testing

In this section, a methodology is proposed that allows testing WSN software on different test platforms. This methodology is based on providing generic test-platform adaptors for individual test platforms. Tests are specified in a unified format. Hence, this methodology allows for:

1. describing testcases valid across multiple test platforms in a common specification supporting testcase design and reusability throughout the development process, and
2. executing the testcases automatically on several test platforms allowing for test portability and comparability.

Before presenting an implementation of the test framework in a feasibility study, details on the methodology are provided.

3.1.1 Multi-platform test framework

The multi-platform testing methodology bases on the idea to integrate self-testing capabilities into tests. Self-testing is achieved by explicitly specifying the inputs and the expected outputs of the software under test in an executable format. This enables full automation of the test procedure. Automated testing promotes frequent execution of the tests and enables the developer to continuously compare the implementation's behavior with the specification.

A test framework can utilize such a methodology to employ a unified testcase as shown in Fig. 3.1. The same testcase can be executed on different test platforms: real hardware, such as a testbed, or a simulator. The simulation can target different levels of abstraction, e.g., pure functional simulation, functional simulation with a refined radio channel model, cycle-accurate execution on an instruction-set simulator, etc. The framework does neither implement nor imply the capability to automatically translate a system or its model across abstraction levels. Instead, the framework provides the mechanism, so-called test platform adaptors, that allow testcases that are valid on different abstraction levels to be executed on the respective platforms. Note that these test-platform adaptors need to be created only once for a given platform. However, these adaptors rely on functionality provided by the test platform and wrap this common functionality such as logging some information into a unified interface. The resulting reusability of the application-specific test specification promotes multi-platform testing.

The framework uses a distinction of the System Under Test (SUT), *drivers* stimulating the SUT, *monitors* observing outputs of the execution and *checkers*, which analyze and evaluate the execution to determine whether a test satisfied the specification. The test procedure is partitioned into three phases [WPBT07]: test preparation, test execution, and result checking. In a nutshell, preparation

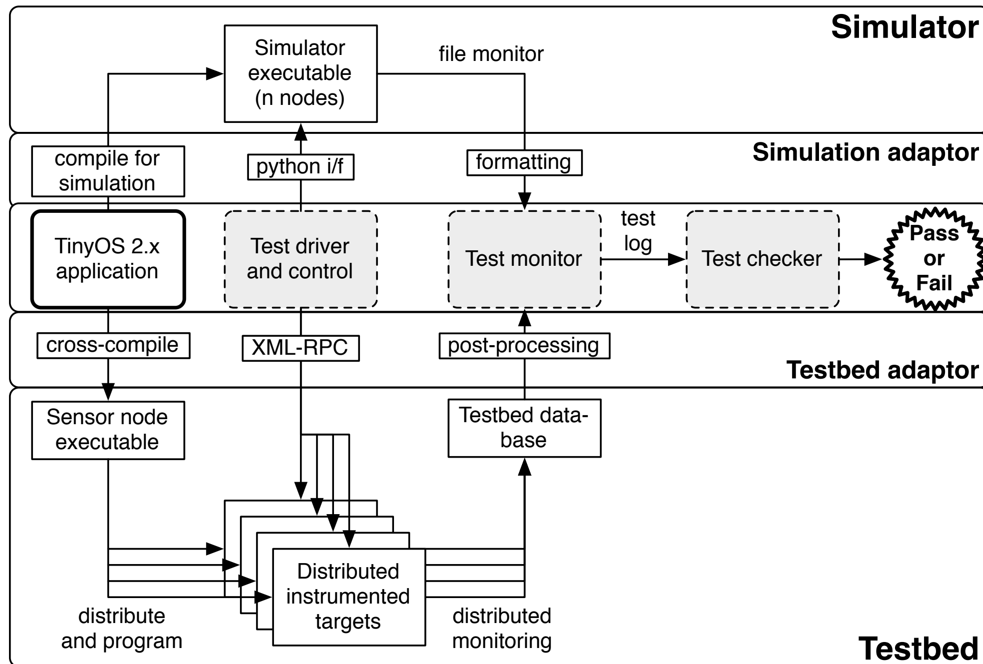

Figure 3.2: Example framework implemented using a testbed and a simulator. Starting point is an application (here TinyOS 2.x) on the left. The common test control logic executes tests either on the simulator or the testbed and collects execution results into a test log in a common format. Test platform adaptors (annotated in grey with a dashed outline) provide a common interface to a test platform for the test control components. The checker on the right determines the final verdict whether the test passed some conditions.

initializes the test platforms and performs instrumentation of the SUT. During the actual execution each SUT is stimulated by a driver and execution is monitored. In the result checking phase, the information logged by the test monitors is collected, post-processed and evaluated. Result checking is the focus of the Rupeas language presented in the second part of this chapter.

3.1.2 Feasibility Study

For studying the feasibility of the proposed framework, two commonly used platforms for WSN testing, a standard WSN simulator and a testbed are selected. For the simulation a code simulator for TinyOS applications is chosen: TOSSIM [LLWC03]. It is a prominent example of an event-driven simulator allowing to use the actual TinyOS application code. For the testbed platform, the Deployment Support Network (DSN)[DBT+07] is selected using Tmote Sky Sensor nodes.

Figure 3.2 depicts the test flow for the example application with a unified testcase, and the corresponding tool chain for the two test platforms including the test-platform adaptors. The starting point of the test flow is a TinyOS 2

application. In a pre-processing step, the generic test primitives are replaced by platform-specific test framework interfaces. The driver for the simulation communicates with TOSSIM via the built-in python interface and generates monitor output in a common format to a log file. The testbed infrastructure provides a service that the test driver uses to communicate to the sensor nodes via an XML-RPC interface. A TinyOS-based test monitor is developed as part of the testbed adaptor to probe variables and state changes via its Universal Asynchronous Receiver/Transmitter (UART) interface. A database collects the distributed monitor information from the DSN and thus provides a common data access point for post-processing. The common output format for both DSN and TOSSIM allows testcases to use the same checker for both test platforms. The checker outputs a pass or fail notification based on the previously specified requirement.

3.1.3 Testcase

The testcase for the feasibility study checks the data yield of the Multihop-Oscilloscope application (cf. Sec. 2.2). In this testcase, the application additionally includes a dissemination protocol that allows the sink to broadcast updates to all nodes in the network. In particular, it defines 10 Tmote Sky sensor nodes set up in a typical office environment. The TOSSIM radio model parameters reflect the topology and communication characteristics of this environment. The parameter fitting is based on data from previous link quality measurements for the Tmotes [MRBT08].

At the beginning of the test, the test driver powers on all target nodes. A local enable flag controls the generation of measurement packets, i.e., it determines whether a sensor node sends its measurements to the sink. Initially, the flag is disabled, i.e., no measurements are sent. After an idle interval, allowing CTP to setup the routing tree, the test driver sets the enable flag on the sink node and nodes send their local measurements. Sensor data is collected for a predefined duration. Subsequently, the enable flag is disabled again and the testcase ends after a phase-out interval for collecting data in transmission. The target monitors log the following information, each with a timestamp: (i) The sending of data packets including the sensed measurements (ii) packet forwarding to the host via the serial port on the sink node including the actual packet over the serial port, (iii) a boot event including code version and node id, and (iv) enable flag changes. The checker uses the monitor information from item (i) and (ii) above to derive the average data yield. The pass or fail condition defined in this testcase is an average data yield of 90%.

Each testcase ran 40 times on both test platforms. Running a testcase on the simulation platform takes about 0.5 min (on an IBM ThinkPad T42 laptop) whereas a run on the testbed takes approximately 11 min. The test time differences are due to differences in the execution time on the platform and significant

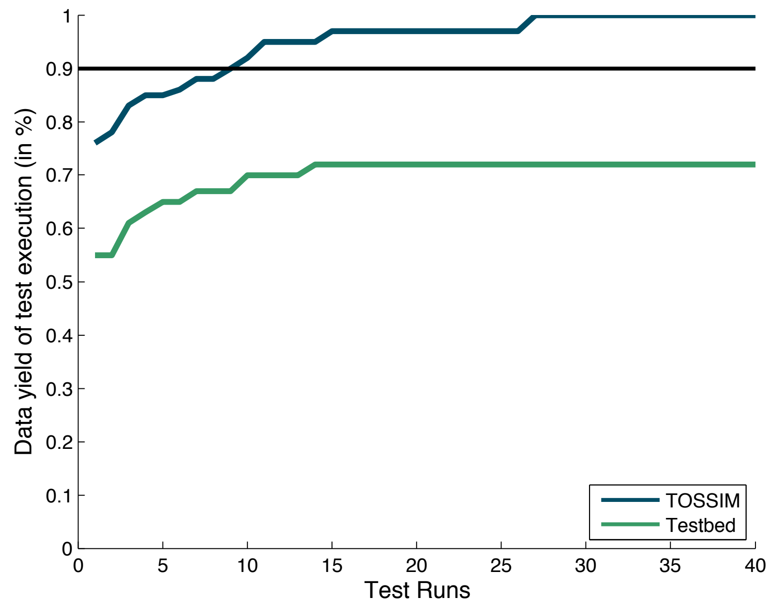

Figure 3.3: 40 independent test runs ordered according to the resulting data yield. 32 out of 40 simulation results passed the test criteria of an average data yield ≥ 90%.

overheads in pre- and post-processing steps in particular code distribution and target programming for the testbed platform. These overheads motivate the aggregation of testcases to test suites in order to mitigate the overhead of these expensive operations.

Figure 3.3 presents the test results. Additional to the binary checker output, indicated by the horizontal line, the average packet yield derived by the checker is displayed. The data in the graph is sorted with respect to the data yield to point out its distribution. While 20% of the TOSSIM testcases fail, no test result on the testbed platform passes the defined criterion. The simulation assesses the data yield overly optimistic. This is due to a simplified modeling of the wireless channel in TOSSIM, e.g., no temporal variability. The distributions of yields for both platforms follow a similar trend but differ significantly in absolute values.

3.1.4 Discussion and related work

A first feasibility study shows that a multi-platform test framework is feasible. By exploiting generic test adaptors with a unified interface for the specification, tests can be ported across different test platforms. This relies on the test platforms providing the same functionality; in this case, a platform adaptor is simply a wrapper around the given functionality. One of the benefits is shown in the feasibility study: Test results are comparable across test platforms and can be used to verify the fidelity of test platform assumptions, here for the

simulator. On the other hand, tests are often designed to be platform-specific, exploiting particular features of a platform, e.g., the scalability of a simulator or measuring actual power consumption of a sensor node. A multi-platform test framework does not provide any benefit for such tests, since they are not portable across platforms.

While this work is the first approach to provide a test framework, two related approaches have been researched that are complementary to the proposed methodology. Zhao et al. [DCG08] discuss a design framework with Que. Their framework enables rapid prototyping and provides some functionality for performance evaluation. Que and the test framework could be combined to provide a comprehensive design and testing tool. Osterlind et al. [ODE+06] present a cross-level simulator. Multi-platform testing could be easily included in such a framework extending the tool into a comprehensive testing tool.

One of the caveats of a multi-platform test framework is the evaluation of test execution logs, i.e., to determine from the monitored information whether a test passed or failed. This is exacerbated when using different node- and test-platforms. In the following, a methodology based on an event abstraction is presented which addresses this problem in detail.

3.2 Analyzing test executions

A crucial part of a test is the analysis of the information monitored during the execution of a system. However, this information is dependent on: (i) the specific SUT, (ii) the test monitors and (iii) the test platform used. While the latter point may be handled by a test framework described above, there is always a dependency on test monitors and so-called logging policies, i.e., which data is logged and its meaning with respect to the application. As an example the relation between send and receive events may be studied. A transmission typically associates a sent log message on a node and a received log message on the intended receiver. However, depending on the logging policy, i.e., where the sending of the message is monitored, the resulting log differs considerably. Logging on the link layer results in sent messages, which have no corresponding received messages, since a transmission may fail. On the network layer a message is only logged as sent if the message has actually been received on the receiver node, i.e., there is a one-to-one correspondence. On the transport layer only the final destination node logs a receive indication. Hence, the analysis needs to consider the logging policy and be able to describe the semantic information of logged information. Since tools and languages need to adapt to the employed logging policy, in practice ad-hoc scripts are often used for analyzing execution logs. However, for increasing reusability of analyses across different tests and test platforms, a more rigorous approach is needed.

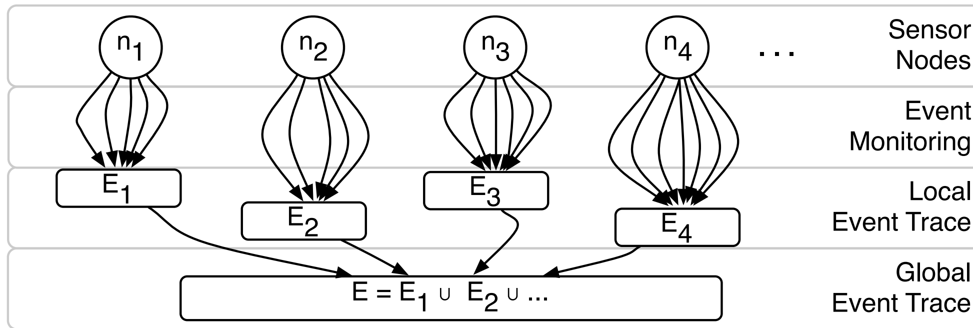

Figure 3.4: Data is logged from individual nodes n_i into a totally-ordered local event trace E_i. These traces are joined into a global event trace E. Since logging from the nodes is concurrent, events in E are partially ordered.

The remaining of this chapter presents a novel programming language Rupeas that allows users to analyze execution traces of WSN systems. This section presents the underlying event framework, which provides the following contributions:

1. is a generic framework for the analysis of WSN logs allowing independent of logging policies, test and test platforms and
2. provides flexible domain-specific operators to describe application-, test-, and platform-specific analyses.

Before presenting Rupeas in Sec. 3.3, this section describes the event abstraction and the corresponding operators on event sets. The event abstraction allows Rupeas to be independent of the monitoring and trace formats and is reusable across projects and platforms.

3.2.1 Traces, Events and Event Analysis

Events symbolize instantaneous information provided by the test monitors during execution. An event e is defined as a single instance produced by a test monitor on one of the k sensor nodes in the system, which are denoted as nodes $n_i, i = 1 \dots k$. As such, events may be test-, application- and platform-specific and hence must be variable in their structure. Examples of events include state changes induced by local code execution such as variable assignments and function calls, state changes induced by external events such as messages either sent or received and (periodic) snapshots of current state.

An *event* e is a k-tuple ($k \geq 2$) of key-value pairs, which minimally comprises a node identifier signifying the origin of the event and a type identifier to classify the event. Further attributes of an event are appended as additional key-value-pairs.

$$e = (node : node_{id},\ type : type_{id},\ key_1 : value_1, \ldots)$$

The notation $e.key_1$ denotes that event e features a key named key_1 and that the corresponding value is: $value_1 := e.key_1$. Events of a single type have a consistent format. As an example, a reboot event logged on node n_{10} at time $t = 10789$ may be represented by the 3-tuple $(node : n_{10},\ type : reboot,\ time : 10789)$

Events are logged on all instrumented nodes. The collection of events from node n_i is a local event trace E_i, which is the set of all events that occurred on the node during test execution. Figure 3.4 shows the process of collecting events from individual nodes and comprising them into the global event trace. The global trace E is the union of the collected event sets $E = \bigcup_{i=1}^{k} E_i$.[1] Local event traces are totally ordered by the (sequential) execution of tasks on each sensor node. However, sensor nodes operate concurrently and only synchronize sporadically, e.g., through message exchange. Hence, events from different sensor nodes may be concurrent, which means that the global event trace is partially ordered. Order information may be included in the event attributes such as recorded timestamps in case a global clock is available.

3.2.2 Event Analysis

A detailed analysis of a large event set is tedious and error-prone. Event analysis alleviates this problem by allowing a systematic approach: By iteratively processing event sets, behavioral aspects of a system are extracted from the global event trace. Figure 3.5 shows an example for this process considering an analysis of all paths of routed packets. Starting from the event set E as the input to the analysis, first the communication events are filtered into a set C. For each event a hop count key is added (with initial value 0), resulting in a set CH. In turn, the event set is processed to yield a set of routing paths RP, from which the set of successful routes SP and the set of failed routes FP can be determined. The outputs of the analysis are such processed event sets, which provide extracted behavioral information. Each processing step uses a specific set operator.

Figure 3.5 also shows that additional processing outside of the event analysis framework may provide debugging and performance information such as the average hop count of successfully routed packets. This can be achieved by embedding event analysis into a scripting language.

3.2.3 Event analysis operators

Event analysis processes events in event sets to extract behavioral information. It offers typical set operators such as union, intersection and relative

[1] Sets are denoted with upper-case letters, while events are denoted with lower-case letters.

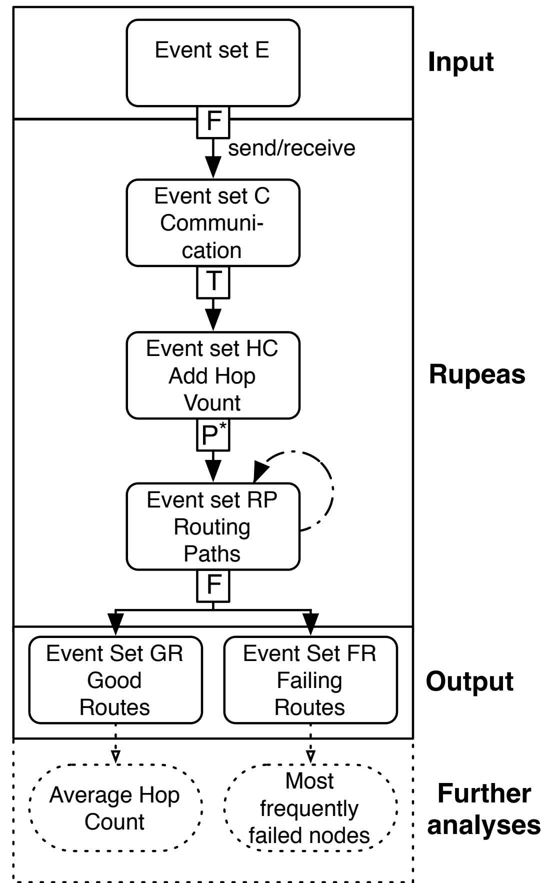

Figure 3.5: Event Analysis: From a logged event set E specific behaviors of a test execution are extracted, e.g., the good routes GR and the failed routes FR.

complement. Additionally, event analysis offers three domain-specific operators especially tailored for processing WSN event sets. Each of the operators specifies which events in the base set it processes using a *selection predicate* φ. Selected event are processed using a *transformation function* f. In the following description, S denotes the input set for the operators. R is the result set, i.e., the output of a given operator.

Definition 3.1 (Selection Predicate)**.** A selection predicate $\varphi(A), A \subseteq S$ is defined on a subset of events and uses relations on the values of specified event keys for selecting events from a base set S.

In a nutshell, predicates select events based on their intrinsic information. In order to select a reboot event from the event trace, the following predicate is used: $\varphi(\{s\}) := s.type = "reboot"$
Predicates may also be defined on multiple events, e.g., to define a send-receive relation: $\varphi(\{s, r\}) := s.destination = r.nodeid \land s.type = "send" \land r.type = "receive"$

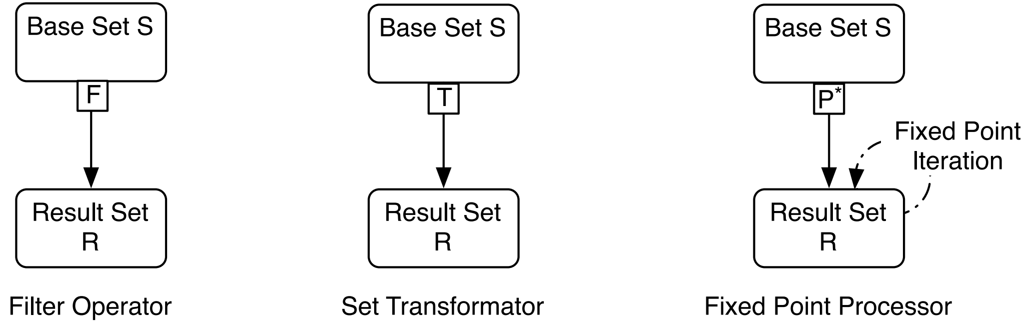

Figure 3.6: The three event analysis operators process an event set S into a resulting set R.

Definition 3.2 (Transformation function)**.** A transformation function is a function $f : 2^S \rightarrow 2^S$ on an event set $A \in 2^S$ returning an event set A'.

A transformation function is used to either (1) add information to events in A or to (2) merge the events in A into a compound event. As an example, to join a send and its corresponding receive event into a compound event (e.g., e_{11} and e_{21} in Fig. 3.7), a transmission, the event is transformed to maintain information from both events $a_1 = (node : n_i, type : "send", sendid : integer)$ and $a_2 = (node : n_j, type : "receive")$:
$g(A) = \{e|a_1, a_2 \in A \wedge e.node = a_2.node \wedge e.type = "transmission" \wedge e.sender = a_1.node \wedge e.receiver = a_2.node \wedge e.sendid = a_1.sendid\}$
The three domain-specific event analysis operators are defined as follows:

Definition 3.3 (Filter operator)**.** The filter operator allows to select a set of events into a subset based on the values of event keys. Filtering is performed on a single set S. The operator returns a single set R, containing the events that satisfy a given predicate $\varphi(s)$.

$$R = \{s|s \in S \wedge \varphi(s)\} \tag{3.1}$$

Definition 3.4 (Set transformator)**.** The set transformator allows to select a subset A from an input set S by using a predicate $\varphi(A)$. Selected events are processed based on a transformation function f. Processed events are added to the result set R.

$$R = \{e|e \in f(A) \wedge A \subseteq S \wedge \varphi(A)\} \tag{3.2}$$

The set transformator operates on a selected subset of events, i.e., one or more.

Definition 3.5 (Fixed point processor)**.** The fixed point processor computes the least fixed point of a given function on an event set and produces a result set. Selection and processing of events in a single iteration is performed as described for the set transformator. Iteratively, the sets R^i are computed, until

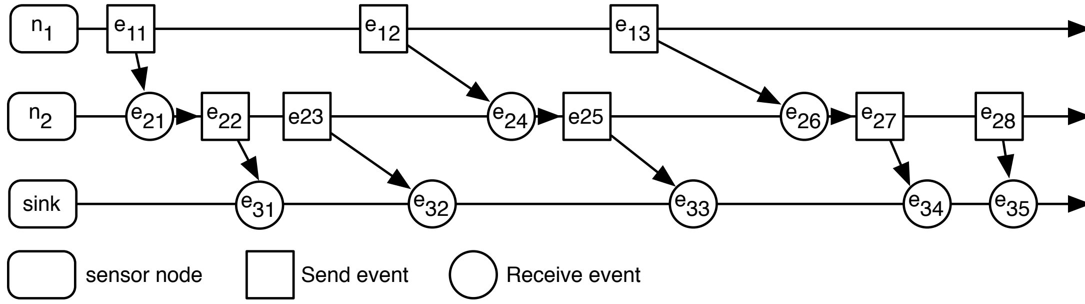

Figure 3.7: Using sending and receiving events to iteratively determine the routing paths. The node where a packet originates is denoted as the *origin*, e.g., node n_1 for the path starting at e_{11}.

a fix point is determined, i.e., $R^k = R^{k-1}$. All events that are not selected by the predicate are maintained for each iteration.

$$\begin{aligned} R^0 =& S \\ R^i =& \{e | e \in f(A) \wedge A \subseteq R^{i-1} \wedge \varphi(A)\} \cup \\ & \{e | \nexists A \subseteq R^{i-1} : e \in A \wedge \varphi(A)\}, i \in \mathbb{N}, i \geq 1 \end{aligned} \tag{3.3}$$

A main application of the fixed point processor is to determine routing paths as shown in Fig. 3.7. Iteratively, send and receive events are joined satisfying a given predicate such as having the same origin and the same sequence number. Fig. 3.6 illustrates the three operators as they are used in the initial example of event analysis for routing analysis shown in Fig. 3.4.

3.3 Rupeas

Rupeas (**Ru**by **p**owered **e**vent **a**nalysi**s**) design goal is to provide a simple, concise notation by using the domain specific abstraction of event analysis. Rupeas provides the following contributions:

1. It enables an automated analysis and checking of event traces.
2. It uses an event abstraction, which allows the formulation of tests to be independent of logging policies and sensor node platforms.
3. Rupeas is a language especially designed for WSN event trace analysis. It facilitates writing analyses using a concise, declarative notation.

Rupeas does not impose any structure on the monitoring and its formats. With merely adding a simple event parser, Rupeas is usable for any project. Even heterogeneous logging is supported as described in Sec. 3.3.3. Moreover,

it is targeted for integration into a larger test (cf. Sec. 3.1) or design [DCG08] framework. Hence, Rupeas needs to be integrated with a scripting language allowing for typical testing tasks such as test automation, data processing and visualization. To this end, Rupeas is integrated with a scripting language: Ruby.

3.3.1 Domain Specific Language

Event analysis is an abstraction using the notion of events and event sets. This domain specific abstraction can be used for formulating a Domain Specific Language (DSL). The basic idea of a DSL is to facilitate the use of a language by focusing on the domain-specific tasks, here processing of event sets, rather than providing a general purpose language. Examples of DSLs include the query language SQL, `make` and `ant` for building software and CSS for web design. DSLs are categorized based on whether they have a custom syntax such as SQL and `make`, or if they are embedded into a host language. Due to its embedding, Rupeas can leverage Ruby's flexible syntax, which facilitates developing internal DSLs with a large degree of freedom on program structure.

Nevertheless, Rupeas is a language specifically designed for the analyzing event sets: it is a DSL with a dedicated purpose. As such, Rupeas leverages the semantic model of event analysis to process event sets. It extracts behavioral information to analyze test executions. Using Rupeas provides a four-fold benefit:

- The event analysis is written in a specifically designed concise language with clear semantics and reduced syntactical noise.
- Supplementary processing and analyses can be formulated in Ruby.
- Rupeas can be easily extended in Ruby, e.g., for plotting results[2] and statistical analysis[3].
- Rupeas is valid Ruby syntax and thus immediately usable on platforms providing Ruby interpreters such as Linux and Mac OS X.

3.3.2 Language implementation

The Rupeas DSL uses a separate context to define the starting point of an event analysis. The Listing below shows the inclusion of the Rupeas library and the creation of the context.

```
require 'Rupeas.rb' #Load Rupeas language definition
Rupeas.new do
#Sandboxed Rupeas context
end
```

[2] http://rgnuplot.sourceforge.net/
[3] http://rubyforge.org/projects/rsruby/

```
Type :senddone do
   with :name=>:seqNo,  :fieldtype=>:periodic, :range=> 0..255
   with :name=>:origin, :fieldtype=>:integer,  :range=> 0..100, :notification=>:error
   with :name=>:dest,   :fieldtype=>:integer,  :range=> 0..100, :notification=>:warning
   with :name=>:nodeid, :fieldtype=>:integer,  :range=> 0..100, :notification=>:error
   with :name=>:time,   :fieldtype=>:float,                     :notification=>:warning
end
     key                 type                   range            notification level
```

Figure 3.8: Declaration of `:senddone` events in Rupeas.

The context provides for a sand-boxed execution of Rupeas scripts, i.e., Rupeas functionality is only valid within the context. The Rupeas library provides amongst other things event and event set classes and the definition of event set operators. The library contains 4 ruby files with ≈ 1300 lines of code.

3.3.2.1 Event Declaration

In Rupeas, users declare each event type in a trace to be analyzed. Rupeas automatically parses the event traces based on the event declaration; the event trace format is a flat file with individual event entries as displayed in Fig. 3.9. Each event lists its properties, with the first column signifying the event type.

Rupeas declares event types in a "`Type`" block: Each property of a type is specified by its name, its type, acceptable values and notification levels on outliers. Types include the basic types (integer, float and strings) with the addition of a "`:periodic`" type, which can be used for wrapping integers, e.g., counters, commonly found in embedded systems, for conversion into a long integer. Notification levels for input trace format problems include a "`:warning`" and an "`:error`". Warnings only display a message and the cause for the warning, while errors stop the analysis. As an example, a send event as shown in Fig. 3.8 specifies that each `:senddone` key is followed by a periodic-type sequence number in the range of 0 to 255. The experiment features 101 sensor nodes. In case the there is an event with a `:nodeid` outside the range 0 . . . 100, Rupeas stops the analysis with an error assertion.

3.3.2.2 Processing event sets

Filtering in Rupeas is as simple as selecting events from a set. Either, events are selected based on having a specific key or by the value of a specific key as shown in Listing 3.1.

```
# select all events having an :origin key
originevents = all[:origin]
# select all events having a :type key with value :senddone
sends = originevents[:type=>:senddone]
```

Listing 3.1: Filtering `:senddone` events from an event set `all` in two steps

...					
51	receive	0	50	51	6.009753
50	senddone	0	50	51	6.009921
44	receive	0	34	44	6.009966
34	senddone	0	34	44	6.010134
62	receive	0	72	62	6.010889
27	receive	0	28	27	6.010983
72	senddone	0	72	62	6.011057
28	senddone	0	28	27	6.011151
...					
:node	*:type*	*:seqNo*	*:origin*	*:dest*	*:time*

Figure 3.9: Event trace excerpt annotated with event keys.

From a set of `all` events only the (communication) events featuring an `:origin` key are selected. Subsequently the events are filtered that feature a `:senddone` type.

The set transformator and the fixed point processor are specified on a given set, e.g., the `all` event set in line 2 of Listing 3.2. In this case the set transformator (`":transform"`) returns a set, which is assigned in this case to the event set `routestart`. It is specified as a block, with the number and name of events it processes, here only `sending`. The selection predicate is specified using two constraints on the sending event. The start of a routing path is selected, hence the event must be a `:senddone` event and its node identifier must match the origin of the packet. If such a route start event is selected, its type gets changed to `:route`. As described before, a fixed point processor is similar to a set transformator but iteratively performs the transformation until a fixed point is reached. Hence, in Rupeas the only difference of a fixed point processor to a set transformator is an `":iterative"` indication on the according code block as shown in line 9 of Listing 3.2.

Selection predicates are differentiated depending on their usage: Constraints (`"constraint"`) are global invariants on the set. *Selections* (`"select"`) specify predicates, which depend on the specifics of the events in the set. Selections only allow for conjunction of terms. Selections are composed by disjunction into a compound predicate.[4] An example for selections is shown in Listing 3.2 for determining the actual route in the `":iterative"` step. The association of individual send and receive events depends on the current events in the set: In Fig. 3.7, e_{11}, a `:senddone` event is first merged with e_{21}, a `:receive` event, forming a `:route`. This `:route` event is subsequently merged with the `:senddone` event e_{22} and so on. Note that in this case, the selection depends on the current state of the processing of the set. As an example of a global invari-

[4]This means that selections are formulated in disjunctive normal form.

```
1  #Setup the start of taken routing paths
2  routestart = all.transform do |sending|
3   constraint sending[:type]==:senddone
4   constraint sending[:origin]==sending[:nodeid]
5   merging :type=> :route, :nodeid=>sending[:nodeid], :seqNo=>sending[:seqNo],
     :origin=>sending[:origin]
6  end
7
8  # Iterate through transmissions and forwarding
9  routes = routestart.transform(:iterative) do |send, recv|
10  constraint send[:origin] == recv[:origin]
11  constraint send[:seqNo] == recv[:seqNo]
12  select send[:dest]==recv[:nodeid] and send[:type]==:route and
     recv[:type]==:receive
13  select recv[:nodeid]==send[:nodeid] and recv[:type]==:route and
     send[:type]==:senddone
14  merging :type=>:route, :nodeid=>recv[:nodeid], :seqNo=>send[:seqNo],
     :origin=>send[:origin], :dest=>send[:dest]
15 end
```

Listing 3.2: Rupeas code for the routing paths analysis. A complete path is represented by its origin and its final destination but not intermediate nodes. A set transformator determines all events that start the path a packet is routed on through the network (lines 2-6). The fixed point processor iteratively builds up the taken path of the packet by associating transmissions across nodes and forwarding on nodes (lines 9-15).

ant, the packets must always feature the same sequence number and origin. Hence, this is specified as a constraint in the `:iterative` step of Listing 3.2.

Transformation functions are restricted to only generate new events, e.g., based on the events selected by the predicate. These new events may either be simply added (in Rupeas by using the keyword "`create`") or replace existing events (using "`merge`"). Transformation functions are specified in terms of resulting events: Each event generated is specified as key-value pairs of the constituent selected events. As an example, Listing 3.2 shows how the start of a packet route is determined. First, the event must indicate a sent packet (`:senddone`, line 3) and secondly, the origin of the event must match its node identifier (line 4). The transformation function maintains the inherent information in the event, but changes the type to `:route` to mark the route start (line 5). The fixed point processor (line 9) associates a send and receive event. Only packets with the same sequence number and origin are joined (lines 10-11). Now a given event is joined either with a new sent event on the same node (line 12) or with a receive event of a node to which a message is sent to (line 13, `sending[:dest] == receive[:nodeid]`). Finally, the events are merged into a `:route` event (line 14), which may be further processed in a further iteration.

3.3.2.3 Implementation details

Rupeas works on large sets of events. Hence, an implementation should be concerned with providing efficient algorithms. An optimization of Rupeas speeds up the execution of operators. Rupeas constraints specify when selection predicates hold globally and can internally perform a partitioning of the set for a considerable speed-up (cf. Listing 3.2).

This optimization is illustrated on the routing paths example depicted in Fig. 3.7. Each taken path has a dedicated start (origin) and some destination. Typically, the path of a packet is uniquely identifiable e.g., by a sequence number, which needs to be the same across all logged events and is reflected in the selection predicate (`constraint sending[:seqNo] == receive[:seqNo]`). For exemplification, it is assumed that there are k nodes each node sending l packets to the sink. Assuming an average hop count of m, each packet incurs and average number of m transmissions (one from the origin node and $m-1$ times forwarded by other nodes). Each transmission incurs 2 logging events, one sending event and one receiving event. Hence, the total number of events n is in the order of $n = k \cdot l \cdot m \cdot 2$. For determining the paths, the fixed point processor is used. In each step of the fixed point processor a sending or a receiving event is joined into the route event (cf. Listing 3.2). However, the fixed point processor iteratively selects any two events from the event set and tries to match them based on the selection predicate. In the worst case, the fixed point processor needs to examine each pair of events before finding a match or stopping if there is no match and hence no change. A naive approach for the fixed point processor on the complete set of transmission requires in such a worst case a substantial number of evaluations: $((k \cdot l \cdot m \cdot 2) - 1)!$.[5] On the other hand, before applying the fixed point processor the set of all transmissions events can be partitioned based on the origin node and the sequence number information in each event. This results in small subsets that only contain events that were sent from the same origin node and feature the same sequence number. The fixed point processor needs to perform considerably fewer operations and builds each of the $k \cdot l$ taken routing path individually: $k \cdot l \cdot ((m \cdot 2) - 1)!$. Note that typically $k >> m$ and $l >> m$, so savings are significant. As a small example for a deployment of 10 nodes each sending 5 messages and approximately 3 transmissions per message, a naive approach would take $149! \approx 4 \cdot 10^{260}$ fixed point processor steps, while the partitioned approach requires $6,000$ steps in the worst case.

[5]The worst case is just used for illustration. The average case will typically be substantially better, however the benefits of the approach remain.

```
Type :senddone do
 with :name =>:nodeid,:fieldtype=> :integer, :range => 0..25
 with :name =>:destination, :fieldtype => :integer, :range => 0..25
 with :name =>:origin, :fieldtype => :integer, :range => 0..25
 with :name =>:seqNo, :fieldtype=> :periodic, :range => 0..255
 with :name =>:ack, :fieldtype => :string
end
Type :received do
 with :name =>:nodeid,:fieldtype=> :integer, :range => 0..25
 with :name =>:destination, :fieldtype => :integer, :range => 0..25
 with :name =>:origin, :fieldtype => :integer, :range => 0..25
 with :name =>:seqNo, :fieldtype=> :periodic, :range => 0..255
end
Type :drift do
 with :name =>:nodeid,:fieldtype=> :integer, :range => 0..25
 with :name =>:neighborid, :fieldtype => :integer, :range => 0..25
 with :name =>:interval, :fieldtype => :integer, :range => 0..25
 with :name =>:absolute_drift, :fieldtype => :float
 with :name =>:drift_in_interval, :fieldtype => :float
end
```

Listing 3.3: Harvester specific events collected on each node (without notification levels).

3.3.3 Case studies

In the following, event analysis is shown on two different examples using the Harvester and the MultihopOscilloscope application (cf. Sec. 2.2).[6]

3.3.3.1 Debugging the Harvester

In the first part of the case study, the Harvester application is tested for Tmote Sky sensor nodes using the Deployment Support Network (DSN) [DBT+07]. Data is forwarded to a single sink. In this case study, heterogeneous logging is performed: the sink node directly logs to a PC via the serial interface, while the logs of all remote nodes are collected via the DSN. Listing 3.3 shows the event types that are produced by the instrumented Harvester application. Receive and send event types are extracted from the TinyOS 2 event handlers for a path and data yield analysis. Measurement events allow for drift analysis as described in this case study. The following analysis focuses on the Harvester with respect to its data yield and its energy-efficiency determining the system lifetime. The events collected for this case study are rather generic. Thus, the case study is representative for a large class of WSN operating systems, sensor nodes and test platforms.

[6] The first part of the case study is performed with EvAnT [WPL+08], the predecessor of Rupeas, which is based on the same event analysis operators. Hence, the results are directly transferable.

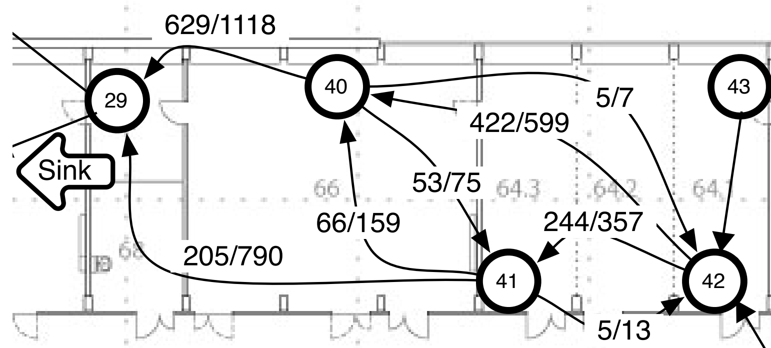

Figure 3.10: Acknowledged and total number of sent packets for selected nodes.

The data yield is determined for multiple experiments performed using 17 nodes, each running for 3 hours. The experiments show that the average data yield for each node is higher than 78%, with nodes in the one-hop neighborhood having an average data yield higher than 90%. A further analysis shows the interesting result, that for the different tests on average each data packet required between 2.6 and 5.2 transmission. For a static network, this number of sent packets should be stable. This indicates a considerable message loss along the routes, which is counteracted by frequent retransmissions. While the effect on the data yield is still tolerable for Harvester, the retransmissions are very expensive with respect to energy. Nevertheless, both results necessitate a deeper analysis of the underlying MAC layer. The analysis of one 17-node experiment took approximately one minute on a MacBook (2GHz Intel Core 2 Duo / 1.5 GB RAM) running OSX 10.5.1 and Python 2.5.1.

A subsequent MAC analysis evaluates the energy-efficiency of the LPL protocol. Figure 3.10 shows results from a 12 hour test run for a selected part of the network. The sink, which is not shown in the figure, is positioned to the left of the nodes. Each link is annotated with the acknowledged and total number of packets sent along this link. In an ideal setting (perfect synchronization and no interference) each packet should be sent only once and should be acknowledged immediately. The discrepancy between the total number of packets and the number of acknowledgments indicates that the synchronization of sender and receiver does not work properly. While interference problems are also possible, previous measurements on the testbed and current test results indicate that this is considerably less likely. Hence, an in-depth analysis of the wakeup time estimation and the time drift detection was performed.

The drift measurement data is computed from measurement packets, which are exchanged by neighboring nodes to determine the clock drift. In a scenario with perfect synchronization for each node pair (n_1, n_2) n_1 measures the same absolute drift of the local timebase to the timebase of its neighbor n_2. That is, if node n_1 determines the drift of node n_2 as $+\tau$, node n_2 will compute a drift of

ID	29	40	41	42
29	-	-0.964	0.932	-
40	0.949	-	0.095	-1.676
41	0.444	0.812	-	-1.012
42	-	-0.201	0.140	-

Table 3.1: Drift in ppm for selected nodes.

$-\tau$. Thus, in a perfectly synchronized scenario the computed drifts add up to 0.

The test is run twice for 24 consecutive hours and the results are accumulated in a single event set. A total of 494,316 events are collected. Table 3.1 shows that certain links are synchronized accurately (e.g., 29 and 40 and nodes not shown in the table), while some links are totally off (40 and 42). Hence, the analysis shows that while the overall data yield is tolerable, the implementation of the synchronized LPL stack needs improvement concerning the drift measurement and interpretation. One of the problems can be attributed to a time-stamping problem of the CC2420 driver as discussed on the TinyOS mailing list [M+07], where corrupted or stale timestamps might be applied to a packet.

3.3.3.2 Routing paths analysis

As another example of an analysis of a data gathering application, Rupeas is applied to data from simulating MultihopOscilloscope with two sink nodes in TOSSIM [LLWC03]. Some of the questions Rupeas can answer are: Which paths were data packets actually sent on? What is the average hop count? Are packets routed equally among the sinks? To this end, CTP is instrumented to log sent and received packets. The simulation topology is a 70 node grid (cf. Fig. 3.11) including two sink nodes (D2, D7). The simulation uses gain and noise models based on USC's Realistic Wireless Link Quality Model and Generator[7]. The log file and input for Rupeas captures data from a 6 hour run resulting in over 2 million events.

The analysis features two basic steps: Loading the event trace using the event description (cf. Fig. 3.8) and using the fixed point processor (cf. Listing 3.2). Rupeas allows for filtering routing paths for final destination, route selection and hop count. The main results obtained are visualized in Fig. 3.11, where all nodes are indexed by their row (A-G) and column (0-9) position[8]: It depicts the yield of each individual node and the sink it mainly sent packets to. The overall yield in simulation is high (99.6%) and the traffic is generally routed evenly among the sinks: sink D7 receives moderately more packets (52.7%).

[7] `http://anrg.usc.edu/www/index.php/Downloads`

[8] As an example, the node where the routes originate from is node B4.

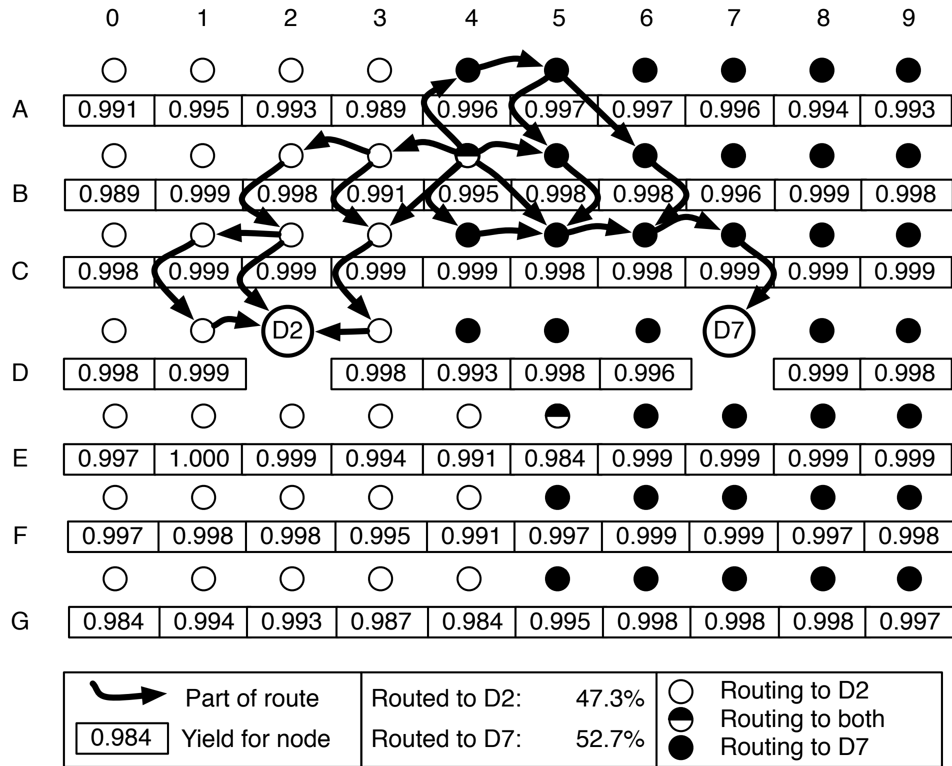

Figure 3.11: Visualization of the routing paths analysis performed with Rupeas.

Most nodes route their packet to a single sink, except node B4 and node E5, which send to both sinks (e.g., node B4 sends 50.4% of its packets to sink D2). The yield for those nodes, especially B4, are nevertheless high. Hop counts for the individual routes differ considerably. For node B4, packets are routed via minimally two and maximally six intermediate hops (average: 5.303). When considering average hop count per origin node, the longest average routes to sink D2 are from node G5 ($\approx$ 6 hops) and to sink D7 from node A8 ($\approx$ 7 hops). Hence, Rupeas shows that CTP provides a high data yield for all nodes and routes traffic evenly among the two sinks in this test scenario.

3.3.4 Discussion and related work

Rupeas allows users to perform common analyses of test executions, e.g., for data collection applications described above: What is the data yield from each node? What do the routing paths look like? Are the routing paths efficient? Such analyses can be performed by Rupeas even for multi-sink systems as shown in Sec. 3.3.3. As an example, routing paths are determined by formulating a relation on send and receive events gathered in traces. Rupeas does not rely on any information about a specific application and test platform, but

rather generates all routing paths by applying user-specified relations on event sets. Further analysis on these productions identifies the desired information, e.g., average routing path length to determine efficiency.

Previously proposed approaches cannot help with such analyses: Diagnostic simulation [MKAG08] using data mining techniques on simulation data, can help you with outlier detection, but does not provide a comprehensive analysis. Note also that outlier detection relies on statistics and learned good behavior for detection, while Rupeas looks at individual executions and determines success based on domain knowledge. As such Rupeas is similar to assertions on distributed, global state [LHL08, RR08]. While event based approaches analyze causal and temporal sequences, i.e., patterns, signifying a specific behavior, such global state-based approaches focus on snapshots of the distributed state, i.e., at a certain instant in time. Hence, these state-based approaches are rather targeted at identifying problems such as selection of a parent of the routing protocol, but cannot help in the analysis of taken routes. Another difference is that with assertions, the collection of information and analysis and oracles are tightly coupled. Even tighter integration is intended with Wringer [TCLS08], a debugging system running on individual nodes, utilizing dynamic instrumentation and collecting global information through in-band communication. At its core a Scheme interpreter is used for evaluating debugging scripts, using predicates to determine localized state conditions. Rupeas takes a different approach: The monitoring and information collection is decoupled allowing for running different analyses on the same monitoring data and analyses from different test platforms.

Health monitoring tools ([RB06, RCK+05]) allow for detecting and debugging failures by providing and communicating additional state information. They extend the communication protocol to provide collaborative, automatic maintenance and recovery of WSNs after deployment. The main target is to provide online health monitoring for typical data collection application in an energy-efficient manner by providing common failure indications. MoMi [dJWL09] provides run-time monitoring support using a model-based diagnosis framework for detecting abnormal conditions during execution. Rupeas' approach is orthogonal, since testing and event analysis is targeted for development, i.e., pre-deployment, and provides an analysis framework for testing various WSN applications on different test platforms.

Analysis tools from other domains, e.g., wired distributed systems, typically have large requirements on the instrumentation of a system. As an example, Pip [RKW+06] is a system for automatically checking the behavior of a distributed system by specifying (or generating) expectations of program behavior and checking the executions. However the considerable requirements on instrumentation prohibit its use in resource- and instrumentation-constrained WSNs. As discussed, these limitations in instrumentation lead to different logging policies and differing communication relations on individual protocol

layers. These application- and logging-specific behavior needs to be integrated in the analysis. Hence, standard methods as used in wired distributed systems cannot be employed. Analysis of wireless systems has focused on inferring information from passively collected information, e.g., in WSNs [RR07, MKL+07] and in WLAN [CBB+06, MRWZ06], rather information is complete, yet dependent on application and test.

3.4 Summary

This chapter presented the concept of multi-platform testing and shows its feasibility on a prototype allowing for testing WSN applications using a simulator and a testbed. Initial results indicate that exploiting test platform adaptors, a specification of the test can be (re-)used across different tools. A major part of specifying tests is the analysis of test executions. Hence, the second part of the chapter presents a novel programming language for analyzing WSN test execution logs. Rupeas exploits an event abstraction and domain-specific operators allowing for a concise syntax in a Domain Specific Language (DSL). By being embedded into a scripting language, Rupeas can be integrated into larger design or test frameworks. Its independence of trace formats and logging policies allows its users to analyze execution logs of different test platforms. Additionally, as shown in a case study, Rupeas enables incremental analyses that can support a developer in debugging the application.

4

Test automation for sensor networks

Test automation is indispensable in software development. Repetitive tasks should be automated to reduce costs and improve the effectiveness of test and analysis. However good tools for the automation process need to be available. Creating such tools is typically time-consuming and expensive. Various approaches and methodologies have been proposed for automated testing of software, both domain-specific such as Selenium[1] for web applications and generic such as autotest[2]. In recent years, Continuous Integration (CI) [PD07] has been proposed in the software engineering community and corresponding frameworks have been developed to support test automation. An open question remains whether such tools can be used for automatically testing WSN systems. Various differences to conventional software testing need to be considered: (i) The quality of sensor network software is mainly determined by its interaction with a dedicated hardware platform. In order to evaluate software using actual executions, tests need to be performed on the actual hardware platform. (ii) Sensor network software is evaluated on different metrics including metrics originating from the interaction with the hardware, e.g., physical parameters such as power consumption. (iii) A test platform needs to provide execution details on these metrics to allow for evaluating the test execution. (iv) The building and evaluation of the software is separated from the execution on the actual test platform.

The first step for creating WSN test automation tools is to develop an architecture that allows for automatically performing tests on sensor nodes in a specific environment. However, executing tests automatically is only beneficial

[1]`http://seleniumhq.org/`
[2]`http://autotest.kernel.org/`

if the results of a test do not need to be manually inspected. Rather tests must be automatically evaluated if they pass or fail with respect to some properties. For functional properties, a language such as Rupeas (cf. Chapter 3) can be employed. However for WSNs, physical properties are of major importance, yet have been neglected as test results up to now.

To this end, a test automation tool for WSNs is presented focusing on power consumption as the prime physical property to be examined. In particular, the following contributions are described:

1. Power Test Architecture (PTA), a test automation architecture, is presented that integrates a CI server with different WSN testing and measurement tools.

2. A novel method for automatically checking power measurements of a sensor node is described.

3. The benefits of using PTA with respect to WSN-specific metrics are shown.

For detailing on WSN specific metrics, the chapter focuses on power consumption of sensor nodes. Hence, the general approach is presented as a PTA with respect to its use for testing power consumption of a sensor node running some software. However, the approach is generic and may be applied for different metrics and for interactions with the sensor node as described in detail in [WBT08].

4.1 Testing power consumption

Energy efficiency is a major concern as WSNs are often battery powered and operated autonomously. For achieving a long term, self-sustained deployment efficient resource usage is of prime concern when it comes to design and realization of wireless sensor networks. The design of a low-power sensor network starts with choosing an appropriate hardware platform. The embedded software needs to provide an energy-aware operation, which requires saving energy by extensively using the hardware's low-power modes. The power-aware programming of such systems is complex and highly error-prone since each hardware component features its own characteristic set of power modes, software tasks can be executed concurrently and hardware components contribute differently to the level of power consumption. Asserting the correctness of such systems includes the assertion of functional and non-functional properties, e.g., such as power consumption as well as real-time properties.

Energy consumption is the integral of the power consumption of the sensor node over time: $E_{total} = \int_0^t P(t)\,dt$. Since also deviations in power consumption need to be detected that manifest themselves only sporadically (in the test),

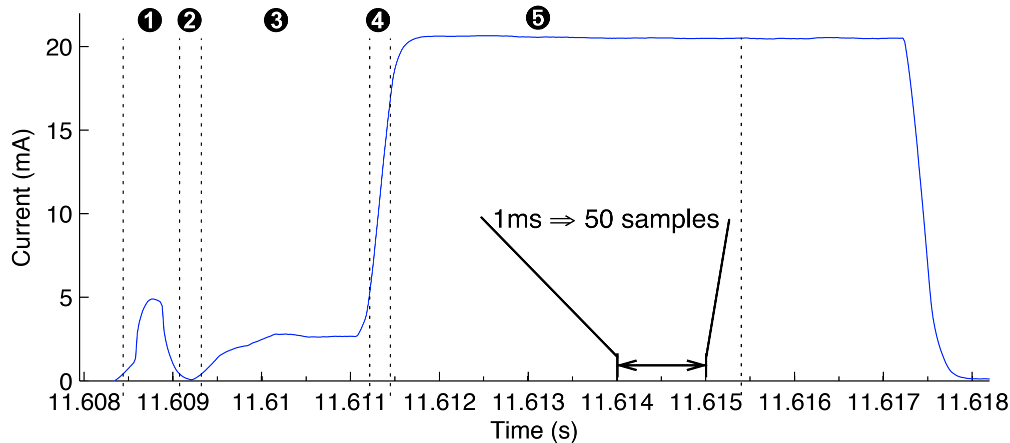

Figure 4.1: Power measurement (current draw under constant supply voltage) taken during the power-up of a sensor node's radio showing different modes: (1) Starting the voltage regulator, (2) waiting for clock startup, (3) starting the crystal oscillator, (4) ramping up the radio, and finally (5) the operational, idle-listening mode.

it does not suffice to look at total energy consumed during a test. Rather an intricate analysis of detailed power consumption measurements over time is required. Such an analysis needs to consider the sleep states of the WSN system as well as duty-cycle patterns to detect even small deviations that may have a substantial effect on energy consumption in a real deployment setting.

Contrary to functional properties, power consumption can be monitored non-intrusively on testbeds [HHP+08], avoiding the probing effects of instrumentation. The obtained power traces capture the power consumption of the system over time, where the patterns in the power consumption may indicate distinctive internal actions. For exemplification one may refer to Fig. 4.1. It shows a passively collected trace of power measurements as taken from the Harvester application (cf. Sec. 2.2) running on a wireless sensor node. The first peak (1) indicates the starting of the radio voltage regulator, followed by the power consumption profile during initialization. The final level of power consumption refers to the system's idle listening state, where the radio is fully powered on and ready for operation, enabling the node to listen on its channel for ongoing traffic.

4.2 Background

Before detailing the proposed architecture for testing power consumption, an overview of the type of errors that occur in WSNs with respect to power consumption and energy-efficient programming, is presented. Furthermore,

the chapter provides some background on *Continuous Integration (CI)*, which is used in PTA.

4.2.1 Error classification

A short overview of different software and hardware errors in the context of low-power embedded systems and energy-efficient programming is presented below, differentiated by hardware-related and software-related errors.

4.2.1.1 Hardware-related errors

Hardware-related errors can be classified as follows:

- *Environmental effects:* An embedded system is deeply integrated into the environment. As such, it is highly susceptible to changes in environmental conditions. One main contributor is temperature and its effects on the clock oscillator [BISV08]. Hence, timing in deviating test conditions may reveal an error due to missing parameterization of test environment conditions.
- *Hardware variance:* hardware properties vary as each component has tolerances on its characteristic properties. Hence, it does not suffice to merely characterize a single node, but rather a representative sample of nodes or rely on the specification provided by the manufacturer. As an example for the effect of hardware variances, Zuniga et al. [Zun04] present the effect on radio communication in wireless sensor nodes. While some hardware variance is intrinsic to each component, highly varying hardware, whether it is sporadically or permanently, must be excluded from deployments as such nodes may drain their battery prematurely and deteriorate system lifetime.

4.2.1.2 Software-related errors

In terms of testing non-functional properties such as power consumption, the most important issues are the effective use of resources, e.g., turning off a component when no meaningful work is performed and returning always to the lowest power state possible. As such sensor nodes sleep most of the time and return to operation by interrupts, e.g., by timers or external interrupts of an Analog-to-Digital Converter (ADC). As such interrupt-related errors, e.g., due to stack overflow, interrupt overflow and missing real-time deadlines [Reg07] are common in low-power embedded software. Moreover sharing resources is complicated by duty-cycling, e.g., of the radio, because contention on resources increases. Other generic implementation errors apply as well, such as control-flow faults due to algorithmic or logical errors and data-flow faults due to initialization, addressing or typing.

4.2.2 Continuous Integration

CI [PD07] is a software engineering methodology that promotes frequent integration, building and testing of the software. It provides rapid feedback to developers. This facilitates early identification of software defects that are typically subject to recent changes. Up to this point, CI focuses on the integration of enterprise-scale software projects designed by large teams and is a common and well known methodology, e.g. in agile development [CLC04].

An overview CI can be seen on the left of Fig. 4.2. A CI server integrates code repositories, tools to compile and build the software, software analysis tools and testing into a comprehensive framework. The overall status of the project, i.e., build, analysis and test results, are visualized, e.g., on a webpage, for communication in the development team. Regular builds in CI are either triggered upon code changes, by user requests or periodically. Builds are referenced to a specific version of the code base in the repository using a unique build id. Test jobs consist of a compilation process that is handled by the CI framework: (i) job formation with subsequent submission to a testbed, (ii) distribution of code to target devices, (iii) synchronous start of all target devices and (iv) log file and power measurements collection. Depending on the context of the test, analysis can take place online, e.g. for the monitoring of operation or in more detail offline after completion of a test job.

Through the integrated approach, execution is greatly simplified and data from all test jobs is logged in a repository that references the actual software code version under test. This assures a maximum of transparency and the ability for a comprehensive post-execution analysis and evaluation.

4.3 The Power Testing Architecture

This section presents PTA for testing WSN software combines established methods from software engineering with WSN-specific tools, i.e., the execution of the software on a WSN platform in a realistic environment and the profiling of power consumption of sensor nodes.

PTA is based on integrating a testbed with measurement devices, e.g., for power consumption, and test control devices, such as a frequency generator, into an off-the-shelf CI framework as shown in Fig. 4.2. It additionally allows the configuration of tests such as specifying a supply voltage and environmental control, e.g., temperature as discussed in [WBT08]. In a particular proof-of-concept setup, a testbed (the DSN [DBT+07]) is integrated with CruiseControl [Cru10], an open-source CI framework. In this work, the focus is on power traces, i.e., measurements of power consumption over time, and their evaluation in a test. Power measurements are performed with an off-the-shelf power analyzer, an Agilent N6705A, providing automation of

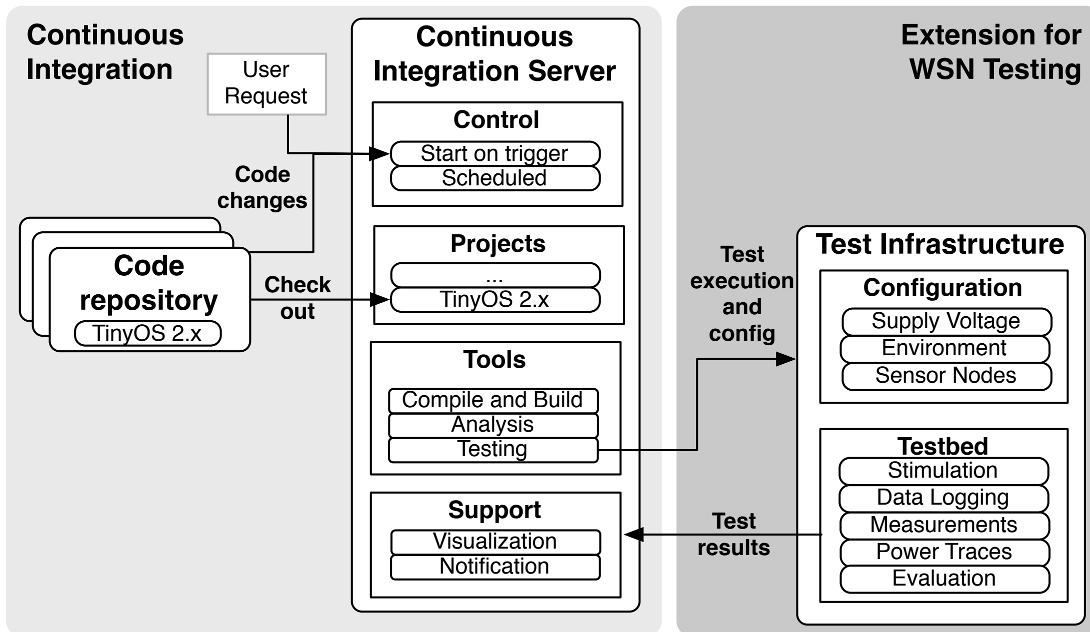

Figure 4.2: A CI server is combined with a WSN specific test infrastructure that provides detailed stimulation and monitoring, including power measurements and the evaluation thereof.

the measurement process and data collection via an Ethernet connection. In this test setting, each sensor node is provided with a stable voltage supply. Hence, only current draw measurements are required for determining power consumption. Measuring power consumption and current draw allows for trending and parameter characterization.

4.3.1 Physical parameter extraction

For detailed characterization of the system performance and especially to pinpoint specific behavior, including variation of the power consumption, traces of power measurements are a vital resource. In order to characterize device variations, but also to understand the interplay of the power supply (battery, regulated power, solar) and the system under varying load conditions, measurements with a fine time resolution are necessary. The finest resolution to consider is the order of the (MHz) clock frequency of the microcontroller. However, sampling at higher frequencies requires measurement equipment with higher fidelity and results in large traces. Additionally, many components such as the radio only change state in the order of milliseconds. In the following the focus is on testing the sensor node and its components rather than the power supply. Hence, a coarser time resolution is chosen (sampling at 20 μs). For a prototypical implementation of the PTA, only a subset of nodes is

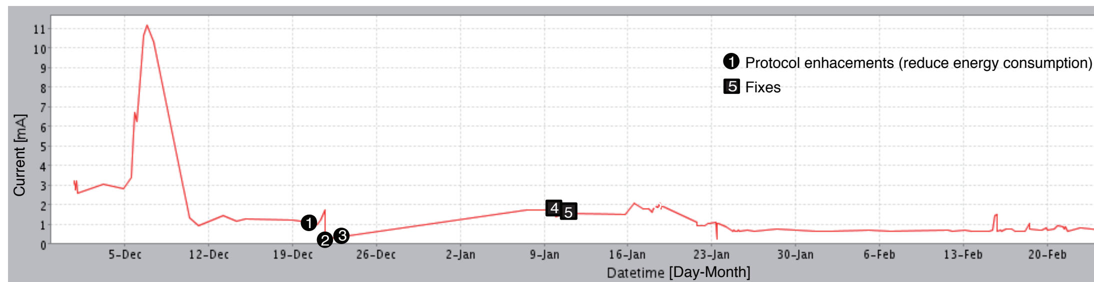

Figure 4.3: Harvester project evolution showing measured average current draw (under constant supply voltage) extracted from CruiseControl. There are various code changes during the project; some examples are marked to indicate specific protocol enhancements (circles) or fixes (rectangles).

instrumented with lab equipment to allow for detailed analysis and automate the process using an interface to CruiseControl.

4.3.2 Cognitive aids

For each build and its associated tests, all data generated, i.e., logs, build artifacts and test results, are stored in a central data structure available through a web based reporting interface [Com08a]. The graphical reporting interface helps to present an overview of the most critical aspects from the wealth of information and contexts. Average power consumption history (Fig. 4.3) is invaluable for analyzing the application's behavior evolution over the development process

As an example consider Harvester [Com08b] (cf. Sec. 2.2). Harvester is designed for long-lasting deployments, thus critically optimizing power consumption. Harvester is in its final development stage, where in-depth optimizations try to minimize power consumption wherever possible. Figure 4.3 shows collected power consumption data from the Harvester application over the development cycle. The visualization provided by the CI framework allows testers to track power consumption changes along the timeline. With the help of such cognitive aids, effects of fixes (in rectangles) and optimizations (in circles) are easily traceable and analyzable.

Change 1 was the initial commit of the synchronized MAC built above the LPL mechanism of the existing TinyOS 2.x CC2420 radio stack. Change 2 on the 22nd of December introduced a better estimation of the clock drift, resulting in a considerable decrease in power consumption, while change 3 enhanced synchronization by discarding of invalid time measurement. A functional enhancement of adding real-valued sensor values and a fix for the Sensirion sensor driver resulted in an increase power consumption on the 9th of January (fix 4). Fix 5 added a CRC check for measurements.

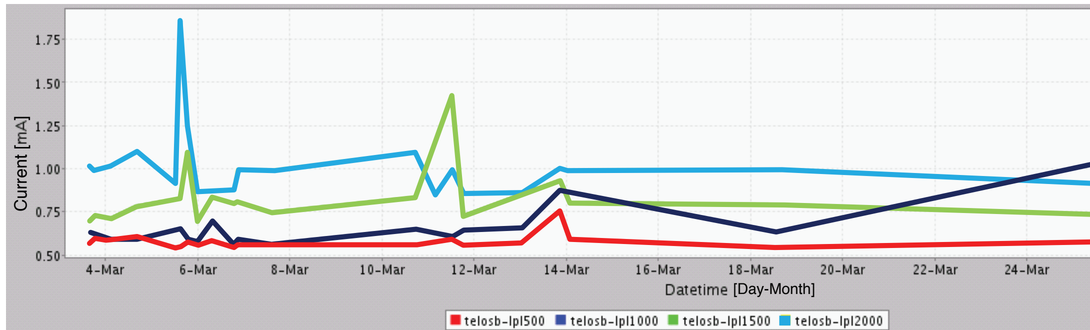

Figure 4.4: Parameterized Harvester project showing measured average current draw (under constant supply voltage) as presented in CruiseControl. Application changes have differing effects for the different parameter settings.

Harvester is tested for each new build on a small network of a base station and two sensing nodes. One of the sensing nodes is monitored. In this setup, the measurement period is configured as 5 minutes. It includes start up conditions, since the initialization phase is an important part in its use for building monitoring. The mean current draw over the whole monitoring window is determined and presented for each build.

Another feature provided by PTA is the ability for parameterized application projects, where the application is automatically built, tested and profiled for a set of differing parameter settings. In the particular example of Harvester, different values for the duty cycle of the LPL protocol are set (T_W, cf. Sec. 2.2.1). For each of the parameters, the average power consumption for a test is recorded and its history displayed as shown in Fig. 4.4. This allows for tracking and comparing the varying effects of changes on the parameterized application. In this example the start-up behavior is considered as described above. Since Harvester's MAC protocol is initially not synchronized, long preambles are sent. Additionally, the network start-up requires a considerable number of broadcast messages (also with long preambles). Hence, in the start-up phase, long wake-up times result in higher average power consumption than shorter ones.

4.4 Formulating tests for power consumption

Having a test architecture for automatic testing and measurements of power consumption, it is fundamental to automatically process the measurements to determine if the power consumption is acceptable for the given test execution. This section presents a method to evaluating traces of power measurements by using a reference (trace) that describes expected power. In a second step, a prototype implementation of so-called *power unit tests* is presented that auto-

matically executes tests on PTA and uses the power measurement evaluation method to generate the test verdict, i.e., whether the test passed or failed.

4.4.1 Reference-based evaluation

The formulation of a *power unit test* is based on a given testcase, i.e., specified inputs and a given application including hardware and software. The testcase is deterministic, i.e., the state of the system during test execution is known. This allows for formulating a *reference function*. A reference may be determined in different ways: using a model, e.g., derived from a specification or by using previous measurements of the testcase on the target device. Reference functions differ in level of detail and accuracy, e.g., physical characterization of the sensor nodes may be incomplete.

A reference function returns the expected power consumption measured on a sensor node. However, measurements include variations due to noise and hardware differences, i.e., manufacturing variations, or changing environmental conditions such as temperature changes. As such a reference is not described as a function, but is given as a region within boundaries. This so-called *acceptance region* is displayed in Fig. 4.5. If the measurements are within the acceptance region the test passes.

4.4.1.1 Determining the acceptance region

Boundaries of the acceptance region are defined based on the reference function and a quantification of uncertainties, e.g., due to hardware variances and measurement inadequacies.

Definition 4.1. A *reference function* is a function $f : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$ of time t of a measured physical quantity, e.g. power consumption. $f(t)$ is a piecewise, typically discontinuous function that is composed of sub-functions $f_i : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$ with discontinuities $t_i \in \mathbb{R}_{\geq 0}$, $i \in \mathbb{N}$ at the sub-interval boundaries of f. In the following, for the sake of presentation the reference function is reduced to represent piece-wise constant power consumption. The definition of the reference can be generalized to allow for fine-granular modeling of power consumption, e.g., to include charge and discharge patterns. The extension of determining acceptance region boundaries for a polynomial reference is straight-forward and hence omitted here.

$$f_i(t) = \begin{cases} a_i, a_i \in \mathbb{R}_{\geq 0} & \text{if } t \in [t_i, t_{i+1}) \\ 0 & \text{if } t \notin [t_i, t_{i+1}) \end{cases}$$

A reference function is hulled by two bounding functions (f^+, f^-) that are the boundaries of the acceptance region. In order to compute these bounding functions, the intermediate upper and lower bound functions $f_{y,i}^{(+/-)}$ need to be

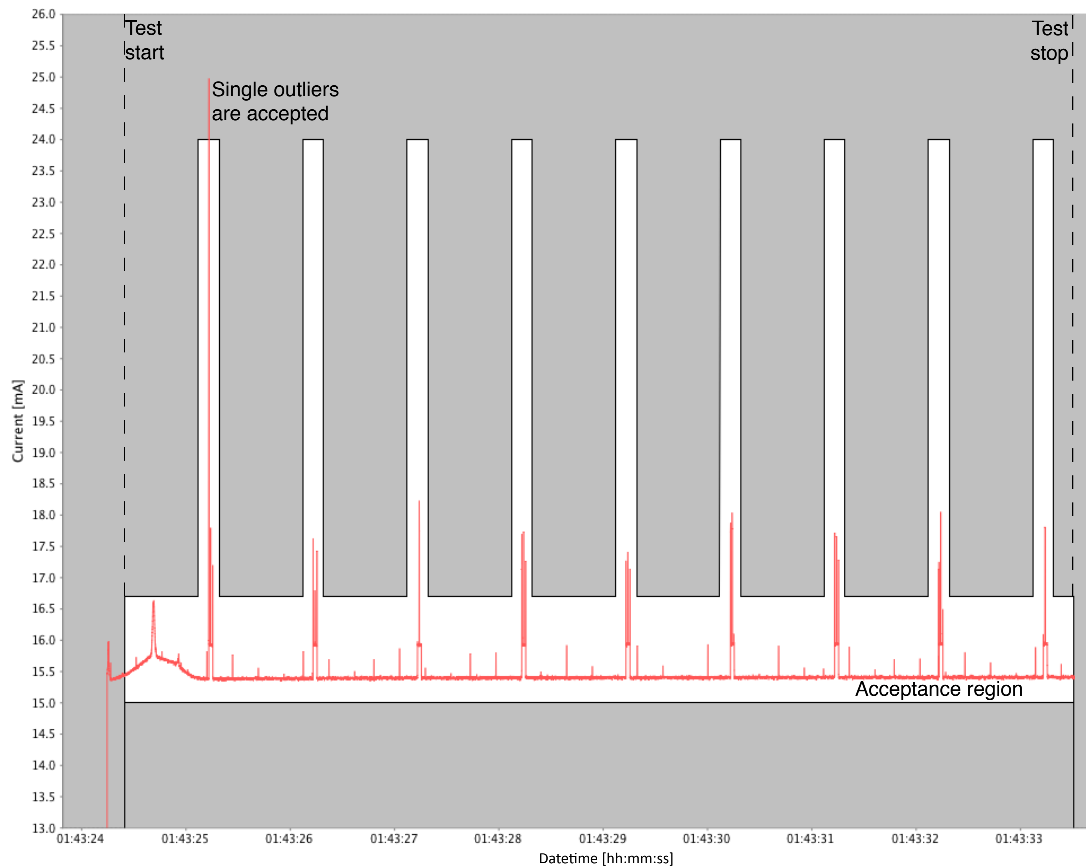

Figure 4.5: Annotated visualization of a power unit test (current draw measurements in mA under constant supply voltage of 3V) for the TinyOS 2.x MultihopOscilloscope application on a TinyNode extracted from CruiseControl. The white area denotes the acceptance region within the reference boundaries. Note that MultihopOscilloscope uses a CSMA MAC protocol and thus never turns of its radio. Hence, the basic power consumption or in this case the current draw of the TinyNode is high, at about 15.5mA.

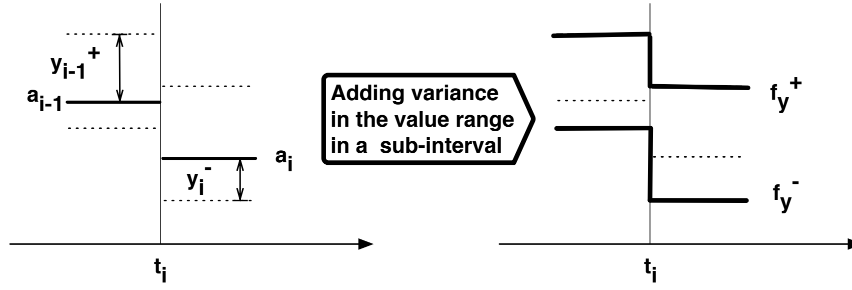

Figure 4.6: Illustration of intermediate bounds derivation given a reference function.

determined first that only account for the variance in value in each interval, but do not consider the sub-interval boundaries t_i.

$$f^-_{y,i}(t) = \begin{cases} a_i - y_i^- & \text{if } t \in [t_i, t_{i+1}) \\ 0 & \text{if } t \notin [t_i, t_{i+1}) \end{cases}$$

$$f^+_{y,i}(t) = \begin{cases} a_i + y_i^+ & \text{if } t \in [t_i, t_{i+1}) \\ 0 & \text{if } t \notin [t_i, t_{i+1}) \end{cases}$$

Variable bounds per interval allow for different granular checking. As an example, a transmitting node has typically larger variation in its power consumption than a node that is in a sleep state. Figure 4.6 illustrates the process of generating the intermediate bounds given the reference and the bound value at the sub-interval boundary t_i. The bound functions $f_y^{(+/-)}$ are defined by additive composition:

$$f_y^+(t) = \sum_{i=1}^n f^+_{y,i}(t)\ ; \quad f_y^-(t) = \sum_{i=1}^n f^-_{y,i}(t)$$

Additionally, uncertainties in time are accounted for using a symmetric variability in time Δt. These uncertainties are small compared to the individual sub-intervals, i.e., $\Delta t < t_{i+1} - t_i, \forall i \in \mathbb{N}$. Thus, the *lower bound* of a reference function is defined as:

$$f^-(t) = \min_{\tau \in [-\Delta t, \Delta t]}(f_y^-(t + \tau))$$
$$f^+(t) = \max_{\tau \in [-\Delta t, \Delta t]}(f_y^+(t + \tau))$$

Figure 4.7 illustrates the effect of this temporal uncertainty around a sub-interval boundary. Hence, f^+ and f^- are the boundaries of the acceptance region used for the power unit test given the intermediate reference and the uncertainty Δt.

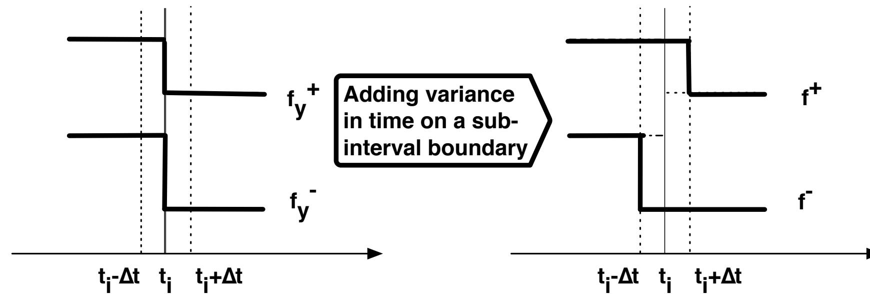

Figure 4.7: Illustration of bounds derivation of the acceptance region given the intermediate bounds and the uncertainty Δt.

4.4.1.2 Checking procedure

Testcases are described using a reference function and a start and stop time, so that the checking procedure may exclude test initialization, e.g., transient effects due to power-on or booting. The measurement apparatus has also some inherent variability. This necessitates an automated compensation in the checking procedure. A temporal shift of the reference function in time is allowed that compensates for test start variances. An offset of the range values (in a limited interval) compensates for differences in the measurement setup. To this end, a least-square analysis between the measured power consumption trace and the reference function is performed. A best fitting reference allows to determine shift and offset to the test-specific reference. The resulting compensated reference function is used to determine the boundaries of the acceptance region. Consequently, for each measurement it is checked that it lies within the acceptance region. If consecutive measurements lie outside this region, an error is asserted. A single measurement outside the acceptance region may be a spurious measurement device artifact as shown in Fig. 4.5.

4.4.2 Power unit tests implementation

As an example for an implementation of a *power unit test*, the MultihopOscilloscope (cf. Sec. 2.2) is tested on a TinyNode 584[3]. The reference is described using an XML specification as depicted in Listing 4.1 for facilitating integration with CruiseControl [Cru10]. It specifies the start and stop time of the checking procedure. The reference function is specified using individual data points to define constant segments, i.e., the value a_i for the interval starting at t_i is specified. The implementation has periodic behavior, i.e., the reference repeats itself after `period` of 1s. In each period in the test, MultihopOscilloscope has 2 distinct segments: (1) for radio idle and (2) for transmitting a measurement. Here current draw is measured under constant voltage supply, e.g., for the first

[3] `www.tinynode.com`

```
<referenceTrace name='MultihopOscilloscope'>
  <start>0.4</start>
  <stop>9.5</stop>
  <xVariance>0.05</xVariance>
  <period>1.0</period>
  <points>
     <point>
       <time>0.0</time>
       <value>15.5</value>
       <yVarianceMinus>0.5</yVarianceMinus>
       <yVariancePlus>1.2</yVariancePlus>
     </point>
     <point>
       <time>0.9</time>
       <value>15.5</value>
       <yVarianceMinus>0.5</yVarianceMinus>
       <yVariancePlus>1.2</yVariancePlus>
     </point>
     <point>
       <time>0.9</time>
       <value>17</value>
       <yVarianceMinus>2</yVarianceMinus>
       <yVariancePlus>7</yVariancePlus>
     </point>
     <point>
       <time>1.0</time>
       <value>17</value>
       <yVarianceMinus>2</yVarianceMinus>
       <yVariancePlus>7</yVariancePlus>
     </point>
  </points>
</referenceTrace>
```

Listing 4.1: XML specification of the reference function of the MultihopOscilloscope application on a TinyNode.

segment the expected current draw is $a_1 = 15.5mA$. The reference specifies variable upper and lower bound values, e.g., $y_1^+ = 1.2mA$ and $y_1^- = 0.5mA$ for the segment. Globally, a time variance (`xVariance`) for a reference function of $\Delta t = 0.05s$ is set. [4]

The resulting boundaries for the reference function are displayed in Fig. 4.5. It also displays the start time at $0.4s$ and the stop time at $9.5s$. Although a single reading is outside the reference bound, no error is asserted, as a single outlier may also be attributed to a measurement artifact. Alternatively, the trace may be filtered before being analyzed.

[4] Note that the format of the XML specification was initially targeted for piecewise linear references, and may be reduced for piecewise constant reference formulations.

4.4.3 Discussion

Creating a power unit test starts with deriving the power consumption for each component of the sensor node, either from a datasheet or using measurements and linear regression as described in [WBLT08] and [FDLS08]. A second step is to determine for a testcase the sequence of different states the system will be in and annotating each state with its power consumption and the time the system resides in the state. This necessitates that a testcase is deterministic. However, non-determinism is inherent in many applications and protocols, e.g., in the form of random backoffs. Additionally, generating a reference for each individual testcase is expensive. Hence, in the following chapter a more generic approach is used: While the concept of bounds on power consumption is maintained, the deterministic reference function is replaced by a (formal) model that allows the specification to include various aspects of the system including non-deterministic or modal behavior. As such a single model comprises a set of reference functions.

4.5 Related work

Related work is divided into two different subjects: Methods for automatically testing for correct power consumption and the infrastructure for gathering information on power consumption. For automatically testing power consumption, this is the first time automatic power consumption is presented for WSNs. There is little related work in different areas. In security research, power consumption traces are exploited for side channel attacks [HMA+08]. The approach is similar: Reference measurements are compared to actual measurements to reason about internal state. In the case of Homma et al., this is only a binary decision: Is a squaring operation or a multiplication performed? In comparison, power unit tests need to identify many internal states, in the order of the number of combinations of each system component's hardware modes.

Concerning test platforms for WSNs, most related work is considering simulation on different abstraction levels and testbeds: A prominent tool for simulation is TOSSIM [LLWC03]. An extension to simulation is tracking the power state of each component during the simulation as performed in PowerTOSSIM [SHrC+04]. Instrumentation of the simulated hardware components and using an additional power tracker, PowerTOSSIM calculates power consumption using models based on measured current draw from Mica2 nodes. AEON [LWG05] provides accurate power consumption prediction for WSN nodes. It is built on top of Avrora [TLP05], a simulation and analysis toolbox for programs written for the AVR microcontroller produced by Atmel and the Mica2, and often used for instruction-level simulation. AEON extends Avrora

with an energy model, similar to PowerTOSSIM's additions to TOSSIM and also uses previous measurements of current draw for hardware characterization of the individual components. Dunkels et al. [DOTH07] discuss online energy monitoring for WSN nodes in the Contiki operating system. The authors instrument device drivers to timestamp activation and deactivation of components and multiply the on-time with an empirically determined current draw. All these methods have a model of power consumption in given hardware modes. In comparison, PTA measures power consumption to assure that such models actually hold for real devices.

Schnyder et al. [SHrC+04] describe in-situ measurements on a testbed: Motelab [WASW05] provides in-situ power measurement for a single node (Mote 118) using a digital multimeter. The timestamped data is available for users in a log file (e.g., for experimental validation). No further analysis support is provided. A tool like SPOT [JDCS07] that provides in-situ measurement of sensor node power and energy allows for large-scale instrumentation of nodes in testbeds allowing for extensive distributed power analysis and testing. Similarly, PowerBench [HHP+08] enables distributed power measurements. PTA provides the groundwork for actually using these platforms to create a distributed, automated approach for testing power consumption.

4.6 Summary

This chapter presents PTA, a testing architecture for automatically testing WSNs. WSN-specific metrics are monitored during the test execution. Measurements can be visualized to give trending information and compare different parameterization. Moreover, measured traces can be automatically checked. To this end, a first approach for power unit tests is presented: By using previous measurements and a piecewise constant model of power consumption, a reference trace can be defined. By allowing for deviation in time and measured value, upper and lower bounds of an acceptance region can be determined, where monitored power traces must be included. If measurements lie outside the acceptance region a test fails.

5

Model-based conformance testing of power consumption

This chapter extends the previously presented testing for power consumption with a model-based approach. As previously described, one of the main challenges for designing and implementing WSNs is the assertion of their correctness, where correctness not only encompasses algorithmic and functional aspects but also extra-functional properties that are closely related to the inherent interaction with the environment. A prime example of an extra-functional property is power consumption. This non-intrusively observable behavior of a WSN may also help in assessing the overall correctness. To this end, this chapter presents a model-based methodology for using non-intrusive power measurements to reason about the correctness of a sensor node.

Power measurement traces are described as timed traces annotated with the measurement values. Visual inspection of those traces or reference measurement-based methods are typically not suitable for the large number of tests that are required when analyzing various properties of the system in different test environments. Rather an automated approach is required. To this end, formal methods can be exploited for the analysis of the system. However, two challenges need to be considered that often hamper exhaustive verification: (1) formal verification may suffer from the state-space explosion problem and (2) measurements have an intrinsic uncertainty due to the measurement instruments, noise, influence from the physical environment and variations in hardware components. The latter point has already been addressed using bounds (on an acceptance region) on power consumption as described in the previous chapter and is utilized for the model-based approach as well.

Model checking techniques have shown to be of great value when it comes to the verification of systems. However, the expansion of all possible system behaviors may yield a large number of system states, which is exponential in the number of a system's concurrently executed activities. This problem, the well-known state-space explosion problem, worsens when timing and other process variables have to be considered. Power measurements from a continuous domain are commonly excluded from any verification process. This is problematic as they are important for guaranteeing lifetime requirements given a system with constrained energy resources. Nevertheless formal methods support an expressive, concise and typically compositional formulation of expected system behaviors. To this end, this chapter presents a scalable approach for formally reasoning about the correctness of WSNs by means of and with respect to power consumption.

The proposed procedure employs formal models for specifying the expected behavior as well as for representing a time series of power measurements. It allows investigating the conformance of expected and observed behavior by using a standard timed model checker. The failure of the conformance tests provides a diagnostic (debugging) trace to the test engineer. This allows the tester to pinpoint implementation errors within a WSN's hardware and software. [1]

As we are dealing with complex hard- and software systems deeply integrated into the environment, the presented approach has to cope with the following challenges:

- As standard instrumentation techniques can falsify the system behavior, non-intrusive techniques are preferable. However with such techniques the internal state of a sensor node is hidden and cannot be observed directly.
- The individual hardware components may contribute differently to the power consumption, but it is only the sum of the individual power consumptions that can be measured.
- The observed power consumption is a result of the interaction between the sensor node and its environment. This interaction needs to be part of the underlying system model and increases the complexity of the conformance test.
- The complexity of systems yields a high-degree of non-determinism ruling out an exhaustive analysis due to the notorious state-space explosion problem.
- As with all physical observations, measurements are subject to uncertainty and measurement inaccuracy.

[1] The proposed approach is also able to consider bidirectional interaction of a system with its physical environment, i.e., the provided measurements serve as input to the system, e.g., for triggering behavior within the system model, or the value of a physical quantity results from the system's behavior.[WLT10]

To deal with these aspects, a non-exhaustive method is devised, i.e., for a time series of measurements and a modeled system, both given as timed automata it is examined if there is a joint execution. One central concept is the mapping of continuous power measurements to a set of distinct, finite intervals in order to (a) reduce the computational complexity of the underlying verification problem and (b) to enable the use of existing tools that are tailored towards value-discrete timed models, such as Uppaal [BY04, BDL04] and TRON [LMN04, HLM+08].

In particular, the conformance between a power trace of a wireless sensor node measured on actual hardware and a specification of the expected behavior of the system is investigated. The main contributions of this chapter can be summarized as follows:

- A new approach for automatic conformance testing wireless sensor nodes is presented that is based on timed automata and model checking.
- An efficient modeling for the composition of physical measurements and a system specification is presented. The chapter discusses optimizations towards computational efficiency. These optimization are required when dealing with measurements from a real system.
- A second approach for the conformance test using an online testing tool is detailed.
- The computational efficiency of both approaches is optimized and compared using a case study of a wireless sensor node application.

The chapter starts with a presentation of the theoretical background. Subsequently, the proposed method is detailed. The main approach uses a model checker for the actual conformance test. A second approach is presented that uses a timed testing tool for the conformance test. Based on a case study using the Harvester Application (cf. Sec. 2.2), both approaches are evaluated and compared. The chapter concludes with a discussion of the proposed method.

5.1 Background Theory

A few notations need to be introduced that are used in the following. First the concept of a timed trace is defined:

Definition 5.1 (Timed trace)**.** A *timed action* is a pair (t, a) where a is some label and $t \in \mathbb{R}_{\geq 0}$ some non-negative time stamp. A *timed trace* $\Pi := (t_1, a_1); (t_2, a_2); \ldots$ is a sequence of timed actions ordered by increasing time stamps, such that $t_i \leq t_{i+1}$ for $i \in \mathbb{N}$.

Timed automata are used as a formal model in the proposed method. In particular, this work focuses on timed automata as used in the Uppaal model checker and follows the notation of the corresponding literature [BY04, BDL04].

Definition 5.2 (Timed automaton extended with variables)**.** A *timed automaton extended with variables* is a tuple $TA = (Loc, Loc_0, Act, C, V, \hookrightarrow, I, AP, L)$ where:

- Loc is a finite set of locations.
- $Loc_0 \subseteq Loc$ is a set of initial locations.
- Act is a set of actions including the internal, unobservable action τ.
- C is a finite set of clocks.
- V is a finite set of (discrete) variables.
- $\hookrightarrow \subseteq Loc \times ClockCons(C) \times VarCons(V) \times Act \times 2^C \times F \times Loc$ is an edge relation, where $ClockCons$ is a set of constraints on clocks and $VarCons$ is a set of constraints on (discrete) variables. These constraints on edges are denoted as guards. F is a set of edge-specific valuation functions on variables.
- I: $Loc \rightarrow ClockCons(C) \times VarCons(V)$ is an invariant-assignment function.
- AP is a finite set of atomic propositions.
- $L : Loc \rightarrow 2^{AP}$ is a labeling function for the locations.

Clock and variable constraints are conjunctions of atomic guards of the form $x \bowtie n, x \in C \cup V, n \in \mathbb{N}_0$ where $\bowtie \in \{<, \leq, >, \geq, =\}$. Clocks are assigned to real values using a *valuation function* $u : C \rightarrow \mathbb{R}_{\geq 0}$. Clocks implicitly increase their value as time progresses. $u + d$ denotes that each clock $x \in C$ is mapped to the value $u(x)+d$, i.e., time is increased by $d \in \mathbb{R}_{\geq 0}$. All clocks in the system increase at the same rate. Clocks can only be inspected or reset denoted by $u' = [r \rightarrow 0]u$, which signifies resetting the clocks in $r \subseteq C$. All other (not resetted) clocks agree with the valuation u, i.e., $u' = u$ for all clocks $C \setminus r$. Variable valuations are determined by an edge-specific valuation function $f_e \in F$ with $f_e : \mathbb{D}^{|V|} \rightarrow \mathbb{D}^{|V|}$, where $\mathbb{D}$ is the finite domain of the discrete variables: $\mathbb{D} \subset \mathbb{N}_0$. Variables are updated on (discrete) transitions, i.e., $v' = f_e(v)$, where v is a vector of variables before the update and v' the corresponding variable vector after the update. Unless specifically indicated, we always refer to a valuation of a clock or discrete variable instead of the variable itself. Furthermore, $u \in g_c, g_c \in ClockCons$ denotes that a clock valuation u satisfies a clock constraint ($v \in g_v, g_v \in VarCons$ is the equivalent for a variable valuation v). We write $(u, v) \in I(l), l \in Loc$ to denote that the valuations of clocks and variables satisfy the location invariant. Note that here a set notation on the valuation is used for the following reason: Invariant and guards are constraints on clocks (or variables) that specify a set of valuations allowed in a given location. Hence, a clock (or variable) valuation is valid if it is included in the set of allowed valuations. This notation is is used for consistency with literature [BY04, BDL04]. AP is a finite set of atomic propositions for labeling locations and $L : Loc \rightarrow 2^{AP}$ is a function that assigns atomic propositions to locations.

The active location of a (single) timed automaton is the location where the execution of the automaton (currently) resides in. A state in a timed automaton is defined by an active location and clock and variable valuations $\langle l, u, v \rangle$. The transition relation $\mathcal{T}$ can be defined as follows:

Definition 5.3 (Transition relation $\mathcal{T}$)**.** A transition relation for TA can either be a delay transition, i.e., the timed automaton stays in a given location and time passes, or a discrete transition, i.e., the timed automaton changes location and updates clocks and variables.

- *delay transition*

$$\langle l, u, v\rangle \xrightarrow{d} \langle l, u + d, v\rangle \text{ if } \forall d' \in \mathbb{R}_{\geq 0} : 0 \leq d' \leq d \Rightarrow (u + d', v) \in I(l)$$

- *discrete transition*

$$\langle l, u, v\rangle \xrightarrow{\alpha} \langle l', u', v'\rangle \text{ if } l \xrightarrow{g_c, g_v, \alpha, r, f_e} l' \text{ and } u \in g_c \text{ and } v \in g_v \text{ and } u' = [r \rightarrow 0]u \text{ and } v' = f_e(v) \text{ and } (u', v') \in I(l'),$$

  where $g_c \in ClockCons(C), g_v \in VarCons(V), \alpha \subseteq Act, r \subseteq C, f_e \in F$ as specified above for TA.

As the delays may be sampled from intervals of $\mathbb{R}_{\geq 0}$, the above transition rules yield an infinite set of state. This infinite set of reachable states is denoted as $\mathcal{R}'$. The infinite set of state-to-state transitions is denoted as $\mathcal{T}'$. These sets can be mapped to the finite quotient system $\mathcal{R}$ and $\mathcal{T}$, yielding the decidability of the reachability question for timed automata; one solely needs to visit the $|\mathcal{R}|$ different system configurations. For further details on clock regions and region graphs the reader may refer to [AD94]. Clock regions and zones are further described in the next chapter.
As the presented approach emphasizes a compositional modeling style, it is required to extend the basic concept to networks of (cooperating) timed automata. In a network of timed automata, the clocks of the individual automata all increase at the same rate. In the following, a brief overview of the related concepts follows; more details can be found in the Uppaal tutorials [BY04, BDL04] or in the textbook [BK08].

- Cooperation via shared variables: Variables can be declared on the level of a network of timed automata, allowing the individual timed automaton to read and manipulate them.
- Rendezvous mechanisms: Uppaal implements different rendezvous mechanisms for jointly traversing over edges within different timed automata. It uses the concepts of channels and signals. By following Uppaal's nomenclature, the terms sender and receiver are used when discussing the synchronization of timed automata via channels: A sender emits a signal on a channel, while receivers react on signals. Senders and receivers differ in their composition: Only a single sender can take part in a synchronization. Senders and receivers also differ in their transition order: Updates on sending edges are performed before updates on

receiving edges. The synchronization of pairs of a sender and receivers is selected non-deterministically, i.e., all possible synchronization pairs are considered in the state space exploration. Note that in the following the traversal of an enabled edge from one location to another location is denoted as *executing an edge*.
The following concepts need to be distinguished:

1. Binary synchronization: One sending and one receiving timed automaton synchronize on the joint transition of dedicated edges: one from the sender, whose edge is labeled by a `channel_id` and an exclamation mark, and one from the receiver, whose edge is labeled by the same `channel_id`, but extended with a question mark. For simplicity, edges are denoted as sending and receiving edges (see the `on!` and `on?`-labeled edges in the timed automata of Fig. 5.5 and 5.6).
2. Broadcast channels: A single sender synchronizes with up to n receivers. This refers to the situation where one timed automaton executes a sending edge, which can be understood as the emission of a signal and where 0 to n receivers execute a receiving edge, which can be interpreted as the instantaneous reception of this broadcast signal. A broadcast requires that each timed automaton that contains (one or more) enabled receiving edges must execute one of these edges.

As some basic terms are needed later, the notions of urgent and initial locations are briefly explained in the following. Fig. 5.5 shows that Uppaal urgent locations are marked with a `'U'` and initial locations are marked with a concentric circle.

- Urgent locations: Within urgent locations no time passes. Thus the system has to execute any of its outgoing edge in zero time once the urgent location is entered. If this is not possible, the execution of the timed automaton deadlocks.
- Initial locations: Each timed automaton in Uppaal has exactly one initial location l_0.

Composed timed automata do not share clocks. Updates of global variables on synchronized edges require special care; they are evaluated sequentially: first the updates on the sending edge are performed, then the one(s) on the receiving edge(s). In case of updates on multiple receiving edges, the resulting update is not well-defined unless the operation is commutative, e.g., incrementing a variable. As the presented conformance testing approach deals with networks of timed automata that are jointly executed, a state of the system can be uniquely defined by a vector of location identifiers, each referring to the

active location held by the respective automaton, and the valuation of all clocks and the vector of values held by the variables. The notation $(\vec{l}, U, V)$ is used for the elements of the set of reachable states $\mathcal{R}$, where $\vec{l}$ refers to the vector of active locations, U refers to the representation of the clock valuations and V to the values currently held by the variables.

5.2 Power Trace Testing (PTT)

The main goal of this chapter is to present Power Trace Testing (PTT). PTT employs a reachability query in order to decide whether a (finite) timed trace of power measurements is included in the traces of a specification of a WSN. For being applicable in an industrial environment the approach is designed in such a way that it can be implemented on the basis of standard real-time tools. The proposed methodology is illustrated in Fig. 5.1: It relies on models of a timed trace of power measurements and a user-defined model that describes the specification of the system under evaluation. Whereas the model of the timed trace of measurements *TM* is automatically derived from the time series of power measurements, the formal model of system behavior *Sys* needs to be (manually) generated from some specification. Having formal models for both, the measurements and the expected system behavior, we can specify a conformance test. Both models are composed into a network of timed automata $Sys||TM$. We formulate the conformance test as a reachability check on the jointly executed network of timed automata:

$$Sys \models TM \Leftrightarrow (\vec{l}, U, V) \in \mathcal{R}_{Sys||TM}, \text{where } \vec{l} \text{ contains } l^f_{TM},$$

where l^f_{TM} denotes the final location of TM and $\mathcal{R}_{Sys||TM}$ is the set of reachable states for $Sys||TM$. Intuitively, l^f_{TM} corresponds to the final measurement in a power trace. In order to automatically determine reachability on the composed model, PTT utilizes a timed model checker. For representing the time series of measurements a suitable model is required, such that (i) the model provides sufficient expressiveness with respect to the tracked power measurements and (ii) the model allows an efficient generation of the set of reachable states of the composed models $\mathcal{R}_{Sys||TM}$. In the following it is shown how timed automata extended with (discrete) variables can be employed for this purpose. The presented approach can be implemented on top of standard timed model checkers such as Uppaal.

5.2.1 Timed automaton models employed in PTT

In the following, the timed automata models for the power measurements (*TM*) and for the system specification are presented (*Sys*).

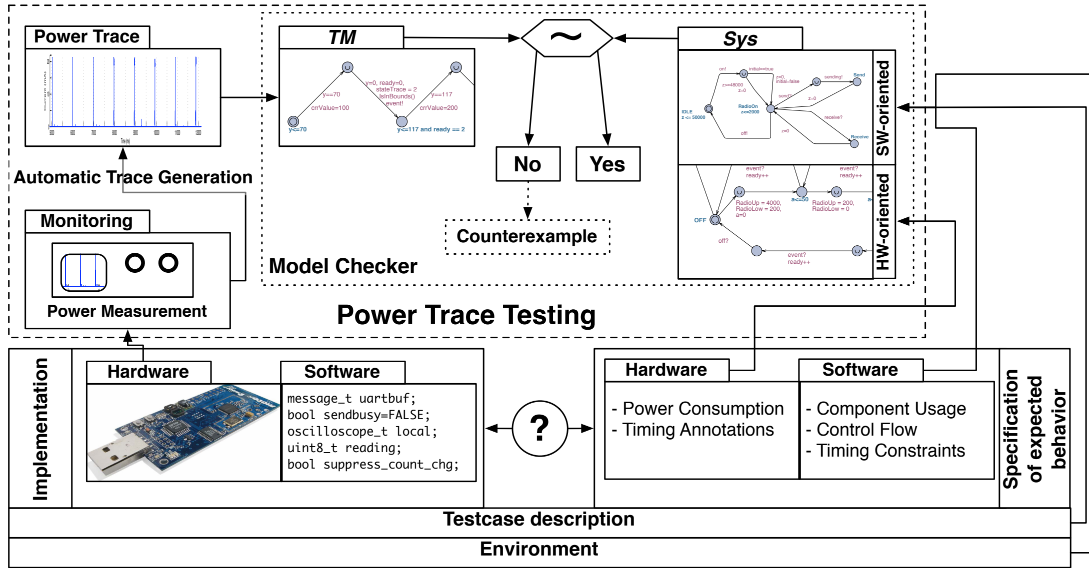

Figure 5.1: Power Trace Testing (PTT) overview: PTT uses (1) a model of the trace of power measurements (*TM*) and (2) a model of the system (*Sys*). *TM* is automatically generated from the trace measured from the implementation. *Sys* is (manually) created from the specification. On the composed model $TM||Sys$, reachability is checked using a model checker. The model checker provides a counterexample if the reachability check fails.

5.2.1.1 Model of timed measurements (*TM*)

The formal model of a time series of measurements is a timed automaton *TM*. It contains a variable p that denotes the measured power consumption and a single clock c_{TM} that denotes the clock valuations at a change of p. Formally *TM* is defined as follows:

$$TM = (Loc_{TM}, Loc^0_{TM}, Act_{TM}, u_{TM}, v_{TM}, \hookrightarrow_{TM}, I_{TM}, AP_{TM}, L_{TM}),$$
$$Act_{TM} = \{\tau\}, u_{TM} = \{c_{TM}\}, v_{TM} = \{p\},$$

where we denote $l_{TM} \in Loc_{TM}$ as a location in *TM*. $l^f_{TM} \in Loc_{TM}$ is the final location of *TM*.

TM is constructed by generating a location in the trace for each pair in the timed trace of measurements. A clock is used to stay in a given location exactly for the time between the current and the previous measurement, i.e., for $(t_1, a_1); (t_2, a_2)$ and assuming the trace starting at time 0, the time in location l_1 is t_1 and for the subsequent location l_2 it is $t_2 - t_1$. However, the size of a trace of measurements is typically very large as (a) typical lab instruments provide a high resolution, e.g., in the order of milliseconds (*ms*) down to nanoseconds (*ns*) for power consumption, and (b) the measurement noise leads to frequent changes of the power level resulting in frequent state transitions. As a result, there is

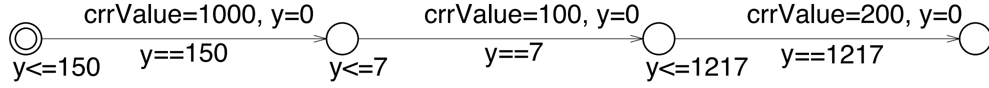

Figure 5.2: A short excerpt of an exemplary *TM* in Uppaal. In this implementation, the power consumption is annotated as `crrValue`. The clock is specified as a clock variable `y`. A value of $crrValue = 1000$ is initially measured for the first 150 time units.

a need for compressing millions of data points into a processable number of locations in *TM*. This is described at the end of this section (cf. Sec. 5.2.3.2), as necessary technical concepts need to be presented first.

5.2.1.2 System specification *Sys*

At first, it is assumed that *Sys* is a single timed automaton. This means that for simplicity of presentation, it is initially ignored that the model may consist of a set of cooperating timed automata. The locations of *Sys* are annotated with invariants on clocks and variables. This results in a description of the system with respect to timing and power consumption. For *Sys*, the following notation is used:

$$Sys = (Loc_s, Loc_s^0, Act_s, u_s, v_s, \hookrightarrow_{Sys}, I_s, AP_s, L_s), \{h_{low}, h_{up}, p\} \subseteq v_s$$

In the following, some aspects of *Sys* are detailed:

1. Set of variables $\{h_{low}, h_{up}, p\} \subseteq v_s$. The model of the trace of power measurements *TM* communicates with the system model *Sys* via a shared variable p that holds the values of the measured power consumption. For dealing with deviations in the measurements, *Sys* employs a pair of variables $h_{low}, h_{up} \in \mathbb{D}$ that specify upper and lower bounds to be respected by the power consumption as represented by variable p, i.e., $p \in [h_{low}, h_{up}]$. Note that *Sys* only reads p; *TM* updates p to signify changes in measured power consumption. The variables h_{low}, h_{up} are manipulated when traversing a respective edge of *Sys*. These bounds on power consumption are equivalent to $y_i^{(+/-)}$ as described in Sec. 4.4.1.1. Note that *Sys* may contain additional variables. However such additional variables are irrelevant for PTT.

2. Set of locations Loc_s. This set consists of two disjoint sets of locations $\mathcal{M}$ and $\mathcal{N}$:

   - *Set of system modes* $\mathcal{M}$
     A location $m \in \mathcal{M}$ represents a mode of operation that possesses a fixed lower and upper bound h_{low}, h_{up} on power consumption p.

Corresponding location invariants allow to invalidate pairs of locations and measured values, namely if $p \notin [h_{low}, h_{up}]$. Hence, it is straight-forward to define location invariants that assert that a provided measurement held by variable p is conformant with the modeled system.[2] The upper and lower bounds are updated when *Sys* executes a transition. In case variable p would be changed in such a way that it does not agree with the currently provided power bounds of a system mode h_{low} and h_{up}, a violation of a location invariant would occur. As this prohibits updates of p, update locations are required. Fig. 5.3 shows a system mode m_1 with $h_{low} = 1$ and $h_{up} = 3$. If the measurement needs to be updated to a value of $p = 9$, this is not possible due to the location invariant. An additional location is needed without an invariant. Subsequently, after the update of p, the system can traverse to a new system mode m_2.

- *Set of update locations* $\mathcal{N}$
  Update locations $n \in \mathcal{N}$ are urgent locations without any invariants. They are artifacts of the presented approach as they allow the update of the shared variable p in another automaton, i.e., by executing a transition of *TM*. Updates on p in system modes may not be possible, as they may violate the location invariant $p \in [h_{low}, h_{up}]$. Hence update location and system mode locations are interleaved.

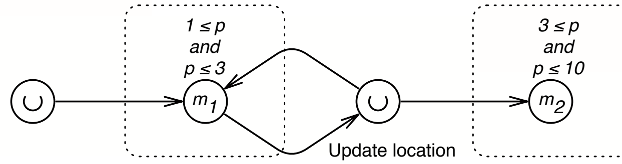

Figure 5.3: The update mechanism of *Sys* as implemented in Uppaal. System modes have an intermediate urgent update location without any invariant.

For formalizing this behavior, the following discrete transition rule in *Sys* is defined:

$$\frac{\langle m, u_s, \{h_{low}, h_{up}, p\}\rangle \xrightarrow{\tau}_{Sys} \langle n, u_s, \{h_{low}, h_{up}, p'\}\rangle \xrightarrow{\alpha_s}_{Sys} \langle m', u'_s, \{h'_{low}, h'_{up}, p'\}\rangle}{\langle m, u_s, \{h_{low}, h_{up}, p\}\rangle \xrightarrow{\alpha_s}_{Sys} \langle m', u'_s, \{h'_{low}, h'_{up}, p'\}\rangle};$$
$$m, m' \in \mathcal{M}; n \in \mathcal{N}; I(n) = (\emptyset, \emptyset); (u_s, \{p, h_{low}, h_{up}\}) \in I(m);$$
$$(u'_s, \{p', h'_{low}, h'_{up}\}) \in I(m'); u'_s = [r_s \to 0]u_s \quad (5.1)$$

[2] As a consequence, the presented approach requires measurements that do not refer to a physical quantity that is an integral over time. It is exactly this assumption that allows the conformance test to stay within the class of timed automata, rather than being forced to make use of linear priced timed automata [BLR05] or other implementations of hybrid automata [Hen96].

where x' denotes a change of x after taking a transition, i.e., either a location change or a change in the valuation. $u_s' = [r_s \rightarrow 0]u_s$ denotes that a subset of clocks $r_s \subseteq u_s$ in Sys may be reset on a transition to a new system mode. Delay transitions in update locations are not possible, since update locations are urgent locations. Delay transitions in system modes follow from the definition of timed automata.

5.2.2 Reachability check for verifying PTT conformance

The trace model TM and the system model Sys are executed concurrently referred to by $TM||Sys$. TM and Sys only cooperate via a shared variable p, i.e., no rendezvous takes place between them, such that $Act_{TM} \cap Act_s = \emptyset, v_{TM} \cap v_s = \{p\}$ holds.
The global variable p is updated when transitions of TM take place. Updates of p may potentially invalidate location invariants of Sys, which is the reason for using update locations in Sys. This specific modeling of Sys results in the transition relations defined below.

5.2.2.1 Definition of transition rules

With the presence of mode and update locations in Sys, the following transition rules are obtained. Note that in order to distinguish transitions in TM and Sys from the transitions in a composed model, a subscript is used on the transition relation: $\hookrightarrow_{TM}$ for TM and $\hookrightarrow_{Sys}$ for Sys. For all transition rules, it holds that $m \in \mathcal{M}$ and $l_{TM} \in Loc_{TM}$.

(a) Delay transition: Sys and TM stay in their current location and increase their clocks with the same duration.

$$\frac{\langle l_{TM}, u_{TM}, \{p\}\rangle \xrightarrow{d}_{TM} \langle l_{TM}, u_{TM} + d, \{p\}\rangle \quad \wedge \quad \langle m, u_s, \{h_{low}, h_{up}, p\}\rangle \xrightarrow{d}_{Sys} \langle m, u_s + d, \{h_{low}, h_{up}, p\}\rangle}{\langle (l_{TM}, m), (u_{TM}, u_s), \{h_{low}, h_{up}, p\}\rangle \xrightarrow{d} \langle (l_{TM}, m), (u_{TM} + d, u_s + d), \{h_{low}, h_{up}, p\}\rangle};$$

with $d \in \mathbb{R}_{\geq 0}$ and
$\forall d' \in \mathbb{R}_{\geq 0}, 0 \leq d' \leq d : (u_{TM} + d', v_{TM}) \in I(l_{TM}) \wedge (u_s + d', v_s) \in I(m)$

(b) Current value of power consumption is accepted by different system locations: Sys may traverse into another system mode that also accepts the current power consumption.

$$\frac{\langle m, u_s, \{h_{low}, h_{up}, p\}\rangle \xrightarrow{\alpha_s}_{Sys} \langle m', u_s', \{h_{low}', h_{up}', p\}\rangle}{\langle (l_{TM}, m), (u_{TM}, u_s), \{h_{low}, h_{up}, p\}\rangle \xrightarrow{\alpha_s} \langle (l_{TM}, m'), (u_{TM}, u_s'), \{h_{low}', h_{up}', p\}\rangle};$$

with $(h_{low} \leq p) \wedge (p \leq h_{up}) \wedge (h_{low}' \leq p) \wedge (p \leq h_{up}')$

(c) Current system location accepts value of power consumption before and after update: *TM* updates the power consumption, yet *Sys* stays in the current system mode, since this system mode accepts the power measurement before and after the update.

$$\frac{\langle l_{TM}, u_{TM}, \{p\}\rangle \xrightarrow{\tau}_{TM} \langle l'_{TM}, u'_{TM}, \{p'\}\rangle}{\langle (l_{TM}, m), (u_{TM}, u_s), \{h_{low}, h_{up}, p\}\rangle \xrightarrow{\tau} \langle (l'_{TM}, m), (u'_{TM}, u_s), \{h_{low}, h_{up}, p'\}\rangle};$$

$$\text{with } u'_{TM} = [r_{TM} \to 0]u_{TM}, (h_{low} \leq p) \wedge (p \leq h_{up}) \wedge (h_{low} \leq p') \wedge (p' \leq h_{up})$$

(d) Mode change: *Sys* needs to enter a new system mode to accept a change in power consumption in *TM*. This will be denoted as a mode change in the following.

$$\frac{\langle l_{TM}, u_{TM}, \{p\}\rangle \xrightarrow{\tau}_{TM} \langle l'_{TM}, u'_{TM}, \{p'\}\rangle \wedge \langle m, u_s, \{h_{low}, h_{up}, p\}\rangle \xrightarrow{\alpha_s}_{Sys} \langle m', u'_s, \{h'_{low}, h'_{up}, p'\}\rangle}{\langle (l_{TM}, m), (u_{TM}, u_s), \{h_{low}, h_{up}, p\}\rangle \xrightarrow{\alpha_s} \langle (l'_{TM}, m'), (u'_{TM}, u'_s), \{h'_{low}, h'_{up}, p'\}\rangle};$$

$$\text{with } u'_{TM} = [r_{TM} \to 0]u_{TM}, (h_{low} \leq p) \wedge (p \leq h_{up}) \wedge (h'_{low} \leq p') \wedge (p' \leq h'_{up})$$

Note that an actual mode change only occurs if it holds that $\neg(h'_{low} \leq p) \wedge \neg(p \leq h'_{up}) \wedge \neg(h_{low} \leq p') \wedge \neg(p' \leq h_{up})$, i.e., the current system mode does not accept the future power consumption and the new system mode does not accept the current power consumption. Our modeling enables a traversal back to the system mode if a mode change is not required. This is significant when *Sys* is composed of a network of timed automata as explained below. Also note that the conjunction of transitions in the antecedent of (a) and (d) denote that these transitions happen at exactly the same time and each transition is only possible in combination with the other. There may be a clock reset on c_{TM} on discrete transitions in *TM*, which is indicated by $u'_{TM} = [r_{TM} \to 0]u_{TM}, r_{TM} \subseteq u_{TM}$. Clock resets on discrete transitions in *Sys* follow the definition in Eq. 5.1.

There is one special consideration for implementing mode changes for a model *Sys* composed of a network of timed automata. Each individual component timed automata model in *Sys* that reads the variable p features mode changes. On a change of p in *TM* only a subset of the component timed automata may require a mode change. Other component timed automata may stay in their given mode. However, all component timed automata must intermittently transfer into an update location to enable a mode change. For this reason there is a back edge from update locations to the corresponding system mode as shown in Fig. 5.3.

5.2.2.2 Implementing the transition rules

Implementing the transition rules within a standard model checker necessitates careful modeling of the update process for p. Figure 5.4 depicts the update process modeled within Uppaal. The model, which could be an excerpt of a system model *Sys*, can unconditionally transfer into an update location; this is needed when the new value of shared variable p necessitates a mode change in *Sys*. At first *Sys* transits from a mode location into an update location. This allows *TM* to execute a transition assigning a new value to p. Once leaving the update location *Sys* updates upper and lower bounds, i.e., assigning new values to h_{low} and h_{up}. Depending on these new values, the location invariants $p \in [h_{low}, h_{up}]$ of potential target mode locations hold (cf. transition rule (d) above) or invalidate the transition. This implies that the overall model $TM||Sys$ deadlocks in an update location if the location invariant $p \in [h_{low}, h_{up}]$ does not hold for any of the potential target mode locations.

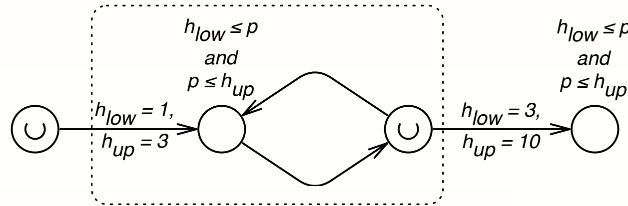

Figure 5.4: The update mechanism of *Sys* as implemented in Uppaal. The dashed box indicates a single system mode and the corresponding update location. The update to power bounds is performed on entering the system mode.

The coupling of *TM* and *Sys* allows to check if the finite timed trace as produced by *TM* is included within the set of traces that can be produced by *Sys*. Using a standard timed model checker like Uppaal, this can be formulated as a reachability problem by querying the reachability of the final location l^f_{TM} of *TM*.

5.2.2.3 Counterexample

If *TM* models a behavior not explained by *Sys*, the concurrently executed model $TM||Sys$ does not reach a state where location l^f_{TM} is marked as active. The last transition that lets a property ultimately fail might be related to the underlying cause similar to the case for liveness checking in [KAJV07]. In the employed reachability property for PTT, this "deadly transition" is given by the last state reached on the longest path of $\mathcal{T}_{TM||Sys}$ (longest with respect to time). A possible method to receive this information is to first label the locations of *TM*, e.g., by indexing them. Subsequently, one can repeatedly check for the reachability of a location based on the label using some search strategy such as binary search.

In order to determine where a power trace fails, iterative calls to Uppaal on reachability of annotated index labels are employed in PTT.

5.2.3 Compositional modeling of the system model

For simplicity, the explanations given above ignored the fact that a system model *Sys* might be built in a compositional manner, where different timed automata represent individual components of the system. In the following, these basic building blocks of the overall system model will be addressed as component timed automata. As an example, a hardware component timed automaton may describe a certain piece of hardware such as a microcontroller with different modes, e.g., performing a computation task or residing in a low-power mode. A software component timed automaton may represent some piece of software controlling some hardware components, where in particular these software components are time-driven. Shared variables or rendezvous mechanisms can be used for coordinating interactions among the component timed automaton as explained in Sec. 5.1. Fig. 5.5 shows that the radio model for a sensor node is periodically turned on for listening on the channel via the label `on`. The corresponding software model is shown in Fig. 5.6: it specifies that the radio must periodically exit its power-off state (with an invariant on the `IDLE` location).

As a major difficulty, such a compositional approach has to cope with the fact that individual hardware components may contribute differently to the power consumption, i.e., each system mode of a hardware component timed automaton may contribute differently. In particular, each hardware component timed automaton has two variables for describing the allowed power consumption in a specific system mode. The overall allowed power consumption is defined by the set of system modes the hardware components of *Sys* are residing in.

As an example, Fig. 5.5 shows the component timed automaton modeling the radio: It features 11 system modes and the corresponding power consumption bounds (here described with the variables `RadioLow` for the lower bound and `RadioUp` for upper bound). Each system mode is annotated with its individual power consumption bounds on the incoming edge. In a powered-off state (`OFF`), power consumption, or equivalently in this case current draw, is lower bounded by $0mA$ and upper bounded by $0.5mA$.

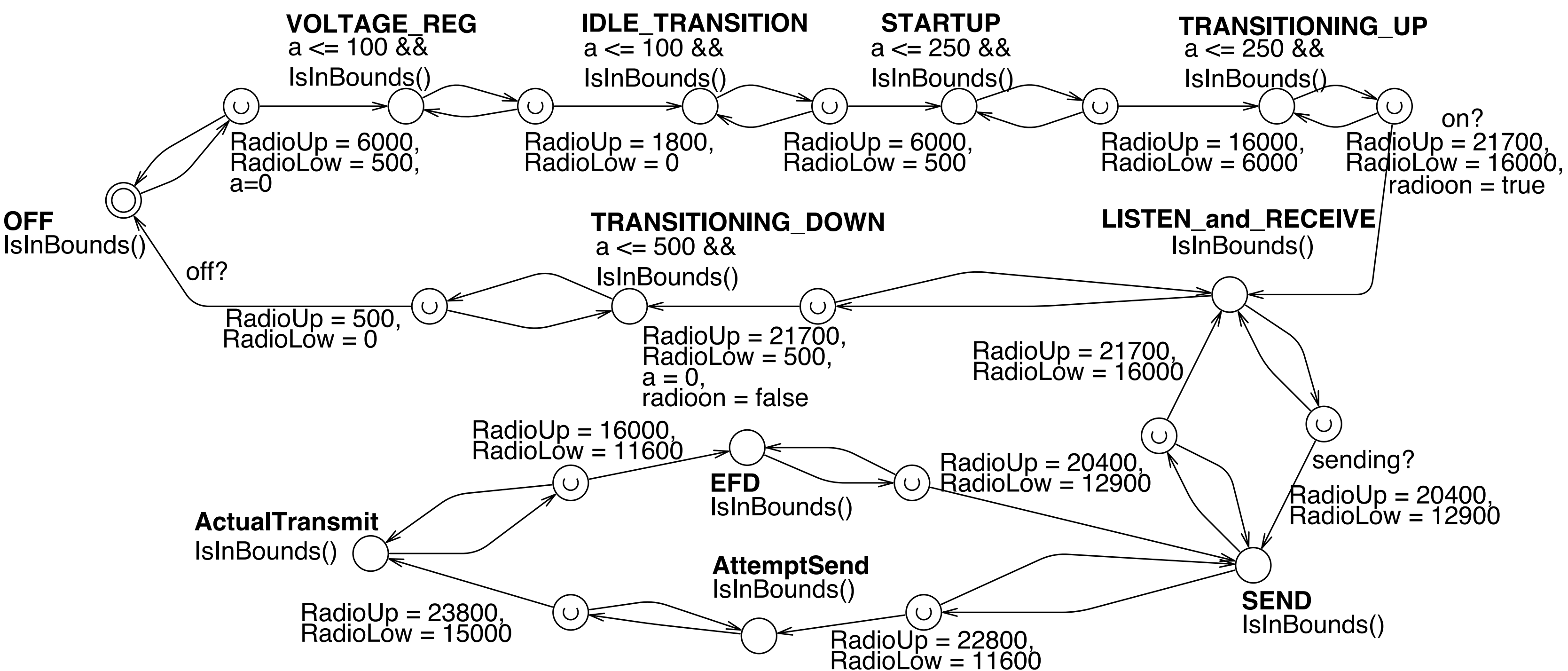

Figure 5.5: Uppaal model of the radio hardware component. Lower and upper bounds on power consumption are annotated as `RadioLow` and `RadioUp` (in μA). A clock `a` controls transition times between system modes. `IsInBounds` is the location invariant on power consumption of the comprehensive system model as described in Listing 5.1.

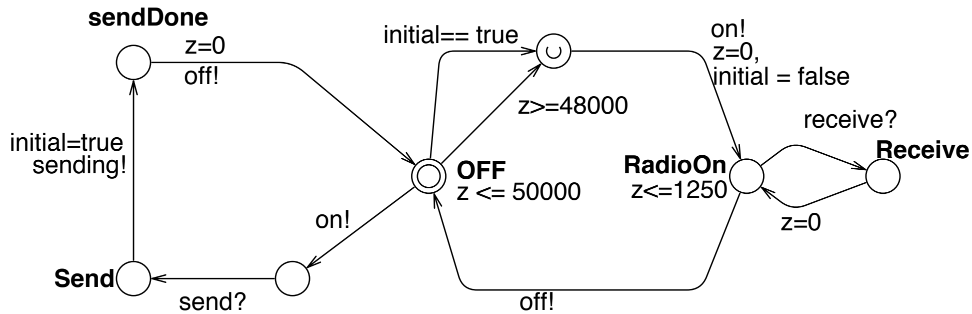

Figure 5.6: Uppaal model of the radio software. The initial state of the software is the `IDLE` location. This location must be periodically exited, at least every 50,000 times units as seen by the invariant on the clock `z: z <= 50000`. The software model uses binary synchronization (channels `on`, `off`, and `sending`) to synchronize with the radio component. Note that `receive?` and `send?` synchronize with a testcase model that specifies when packets may be received and sent.

5.2.3.1 Interval composition

Each component timed automaton i has its own set of variables $\{h^i_{low}, h^i_{up}\}$ that indicate the currently accepted interval. As a system state $(\vec{l}, U, V)$ contains the location of each component, the interval bounds of the overall system model *Sys* can be obtained by adding the lower and upper bounds associated with the location of each n component timed automata: $h^1_{low}, \ldots, h^n_{low}$ and $h^1_{up}, \ldots, h^n_{up}$. The sums are assigned to the global variables h_{low} and h_{up} as introduced above: $h_{low} := \sum_{i=1}^{n} h^i_{low}$ and $h_{up} := \sum_{i=1}^{n} h^i_{up}$. Since the bounds h_{low} and h_{up} are additively composed, the power intervals described by the bounds may not be disjoint. All possible intervals of power consumption can be computed offline given the system model *Sys*.

As an example, the case study in Sec. 5.4 uses two components, a microcontroller as well as a radio. The implementation in Uppaal is displayed below for the bounds of the radio, `RadioLow` and `RadioUp`, and the bounds of the microcontroller, `MCLow` and `MCUp`:

```
bool IsInBounds(){
       int h_up = RadioUp+MCUp;
       int h_low = RadioLow+MCLow;
       if(crrValue > h_up or crrValue < h_low) return(false);
       return(true);
}
```

Listing 5.1: Invariant on system modes checking that measured power consumption (`crrValue`) is included in the specified (additive) bounds.

5.2.3.2 Reducing power trace size

As discussed in Sec. 5.2.1.1, the size of *TM* is a major obstacle. A timed trace obtained from some measurement can include millions of measurements. Given that the input size of models for standard model checkers is constrained, a considerable reduction must be achieved. In order to reduce the number of locations in the sequential *TM*, we can exploit the computed set of intervals of *Sys*: We segment the value range of power measurements into intervals. The use of intervals, i.e., valuations of h_{low} and h_{up}, on power consumption in *Sys* allows PTT to abstract from the measurements. For keeping the number of power intervals as small as possible, PTT exploits the concept of Greatest Common Intervals as described in the following.

5.2.3.3 Greatest Common Intervals

In order to determine a minimal representation of intervals, PTT exploits the concept of a Greatest Common Interval (GCI) partitioning[Str00], whereby a set of non-disjoint intervals, such as the set of bounds H_{Sys}, is transformed into a minimal size set of disjoint intervals H_{GCI}. Since GCIs are disjoint, a single value can be used for the representation of a single GCI: the average value of the interval. Fig. 5.7 exemplifies the procedure when transforming a non-disjoint partitioning of a finite domain into a disjoint partitioning by introducing GCIs. Solid lines indicate the (possibly) non-disjoint intervals, given by different valuations of h_{low} and h_{up}.

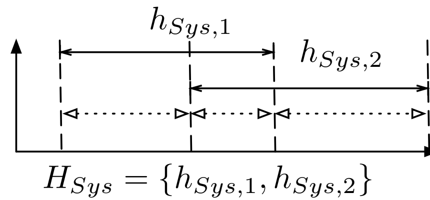

Figure 5.7: $h_{Sys,1}, h_{Sys,2}$ are intervals of two different valuations of the bounds on power consumption h_{low} and h_{up}. For these two non-disjoint intervals, three GCIs are created, i.e., $H_{GCI} = \{h_{GCI,1}, h_{GCI,2}, h_{GCI,3}\}$

The figure shows two intervals $H_{Sys} = \{h_{Sys,1}, h_{Sys,2}\}$. By introducing a set of GCIs the non-disjoint intervals can be transformed into a disjoint partitioning, where the GCIs are indicated by dashed lines, i.e., $H_{GCI} = \{h_{GCI,1}, h_{GCI,2}, h_{GCI,3}\}$. Each of these disjoint intervals represents an equivalence class with respect to power consumption. Different measurements inside a GCI cannot be distinguished. This is used in PTT for reducing the number of locations of *TM*.

5.2.4 Trace Automaton Optimization

The trace reduction described above must preserve the timed behavior with respect to power consumption. The set of GCIs can be computed (offline) from the system specification model *Sys*. As a result, the construction of a compressed trace of measurements is in principle straight-forward. When the compressed trace is constructed, a respective timed automaton can be derived, which is denoted as quotient timed automaton TM' in the following. Below, it will be shown that TM and TM' are equivalent with respect to the timed sequence of GCI-visits, such that

$$Sys \models TM \Leftrightarrow Sys \models TM'$$

5.2.4.1 Quotient transition system generation algorithm

For automating the generation of the quotient timed automaton TM', PTT employs Algorithm 5.1 that creates a list of *locations* l given the measurement samples (*power measurements* p) and the endpoints of the GCIs (*GCIEndpoints*[0...*maxGCI*]). Note that here we assume equidistant measurement samples of power consumption as typically provided by lab instruments. In a nutshell, a location is created for k consecutive measurements when the value of the power measurement is within the same GCI $h_{GCI,l}$. The time passing in a location is determined as $r = k$. The value *crrValue*, representing a GCI, is determined as the (arithmetic) mean value of the GCI endpoints $\frac{1}{2} \cdot (h^{low}_{GCI,l} + h^{up}_{GCI,l})$. Note that for the GCIs of power measurements it holds that $h^{up}_{GCI,l} = h^{low}_{GCI,l+1} \forall l \in \{1, ..., |GCI| - 1\}$.

Fig. 5.8 displays a trace excerpt that shows the GCIs of a typical system model and the resulting GCI-based, compressed trace. Dashed horizontal lines denote the GCI endpoints. The process of generating the compressed trace and its timed automaton-based representation TM', is exemplified in Fig. 5.9.

5.2.4.2 Correctness of quotient timed automaton generation algorithm

Definition 5.4. Each location $l \in TM$ has a distinct residence time $r \in \mathbb{R}_{\geq 0}$ that is given by the time between two differing measurements. This is modeled by an invariant on a location $d \leq r$ with $\langle l, u, v\rangle \xrightarrow{r} \langle l, u + r, v\rangle$ *where* $\forall d \in \mathbb{R}_{\geq 0} : 0 \leq d \leq r \Rightarrow (u + d, v) \in I(l)$. The location invariant and a corresponding guard on the outgoing transition ensure that the transition to the next location has to be taken exactly after r time units have passed.

The example in Fig. 5.9 shows that a typical trace of measurements TM with equidistant power measurements featuring a residence time of one for each location $l \in TM$.

Algorithm 5.1 Generating the locations of TM'. Inputs are the list of *power measurements p* and the list of GCI endpoints *GCIEndpoints*[0...*maxGCI*]. The output is a list of locations l of TM' each indicating the residence time r in that location and the corresponding (abstracted) measurement value *crrValue*.

```
crrValue ← 0, r ← 0, l = [], j ← 0
for all power measurements p do
  i ← 1
  while i < maxGCI && p > GCIEndpoints[i] do
    i ← i + 1
  end while
  if i == maxGCI then
    p′ ← GCIEndpoints[maxGCI]
  else
    p′ ← 0.5 · (GCIEndpoints[i] + GCIEndpoints[i − 1])
  end if
  if j == 0 then
    crrValue ← p′
  end if
  if crrValue == p′ then
    r ← r + 1
  else
    l[j] = new location(crrValue, r)
    crrValue ← p′, r ← 1, j ← j + 1
  end if
end for
l[j] = new location(crrValue, r)
```

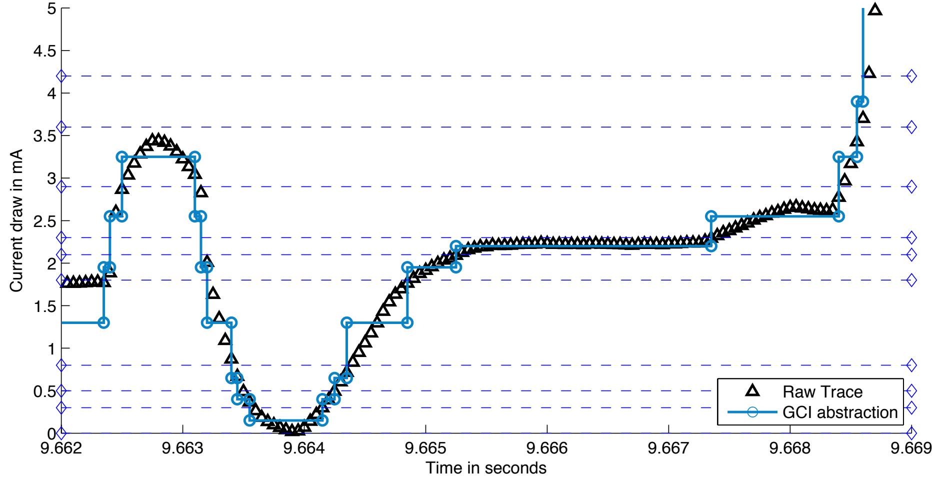

Figure 5.8: Raw measurements of current draw under constant supply, i.e., power consumption, and the compressed power trace using GCI abstraction. Horizontal, dotted lines indicate the GCI boundaries.

Theorem 5.1. The quotient timed automaton generation algorithm (cf. Algorithm 5.1) produces a reduced automaton TM' that is equivalent with respect to the timed sequence of GCI-visits to the automaton TM derived from the original timed trace of measurements.

Proof 5.1 (By induction over the number of sequentially visited states). *Assumption*: A location $l' \in TM'$ is equivalent to a location $l \in TM$ with respect to its inclusion in a GCI iff for a location $l \in TM$ the valuation of p is in a specific GCI h_l, i.e., $p \in h_l, h_l \in H_{GCI}$ and the valuation of p' in the location $l' \in TM'$ lies inside the same interval: $p' \in h_l$. This is denoted as GCI-valuation-equivalent in the following. The automaton TM' is GCI-valuation-equivalent to TM iff there is a GCI-valuation-equivalent sequential location in TM' for each location in TM.

Basis ($i := 0$): Let us assume that we are at a location l_0 of TM that features a power consumption of p_0. The power consumption p_0 is in a given GCI $p_0 \in h_l, h_l \in H_{GCI}$, i.e., the system model *Sys* is indifferent to the specific value within h_l. Hence, l_0 is equivalent with respect to the GCI of the quotient TM' that resides in its initial location l'_0 with $p'_0 \in h_l$.

Inductive step ($i \rightarrow i+1$): Let us assume we are at a location l_i with power consumption p_i with $p_i \in h_l, h_l \in H_{GCI}$. Now there are two possible options for successor states l_{i+1}:

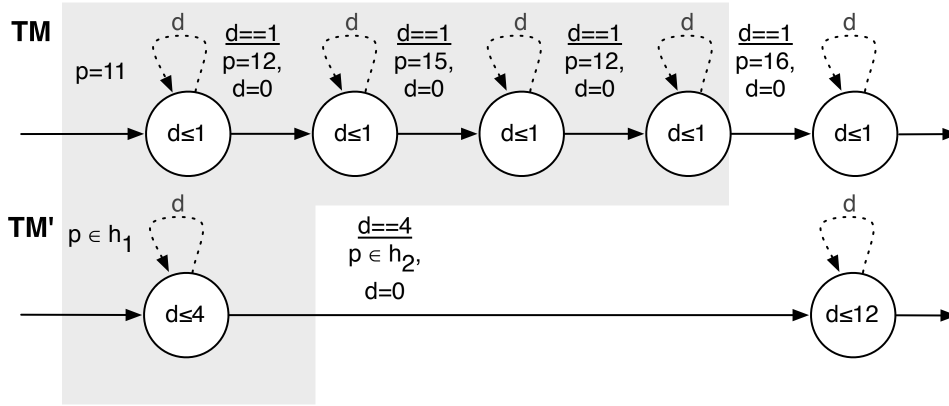

Figure 5.9: Construction of TM' from TM by looking at the GCI-valuation-equivalence. Here $h_1 = [10, 15]$ and $h_2 = [16, 18]$.

1. $p_{i+1} \in h_m, h_m \in H_{GCI}, m \neq l$. A new location l'_{i+1} is generated in TM' with $p'_{i+1} \in h_m$ and residence time $r'_{i+1} = r_{i+1} = 1$. Hence, l'_{i+1} is GCI-valuation-equivalent to l_{i+1} at least for time r'_{i+1}.

2. $p_{i+1} \in h_l$. No additional location is added for TM', but the residence time is increased for l'_i such that $r'_{i+1} = r'_i + r_{i+1} = r'_i + 1$. l'_i features the same valuation with respect to the GCI for both l_i and l_{i+1}. Hence, at each instant in time TM and TM' provide the same valuation with respect to the currently visit GCI.

The induction scheme above proves that we can generate a quotient timed automaton TM' for any time series of measurements. The constructed quotient timed automaton is GCI-valuation-equivalent with respect to the trace model TM for the following reasons: Let TM' be in any location l'_i corresponding to the location sequence $l_p, ..., l_q$ in TM. By construction of TM', it holds that the residence time in l'_i is $r'_i = \sum_{j=p}^{q} r_j$. Note that l'_i is GCI-valuation-equivalent to $l_p, ..., l_q$. Hence it follows that TM and TM' are equivalent for the time t: $\delta_i \leq t \leq \delta_i + r'_i$ with respect to their GCI-valuation, where $\delta_0 = 0$ and $\delta_j = \sum_{j=0}^{i-1} r'_j$. As TM and TM' may only reside in a single location at any period in time, the induction over i covers all points in time of the original time series of measurements and yields again that TM and TM' are GCI-valuation-equivalent.

Lemma 5.1. If TM' is an GCI-valuation-equivalent automaton to TM, then $Sys \models TM \Leftrightarrow Sys \models TM'$ holds.

Proof. It only depends on the timed sequence of GCIs whether the final location l^f_{TM} of trace automaton TM or TM' is reachable within the composed model $Sys||TM$. As this is the same for TM and TM', the lemma holds. □

5.3 Testing power consumption with TRON

As an alternative to the above approach one may also employ a conformance online testing tool such as TRON [LMN04, HLM+08, KGL09]. As this tool also utilizes a timed automata-based system specification, it appears to be highly suited for benchmarking the approach presented so far. A detailed discussion is provided that may help to understand the differences in terms of analysis performance.

5.3.1 Timed input/output conformance relation

TRON is a model-based tool that tests the conformance of an implementation and its timed automata-based specification, where conformance is tested with respect to timed input/output behavior. Analogously to PTT, TRON employs timed automata for specifying the desired behavior of an implementation, i.e., only behavior that is allowed by the specification may be seen in any test run. TRON uses a relativized timed input/output conformance relation `rtioco` introduced by Larsen et al. [LMN04] as defined below. Note that in this section the notations of the original publication are used. In particular, Larsen et al. [LMN04] use a different notation for timed traces; in their work, and hence in the following explanations, a timed trace is a sequence of labels. The labels may include actions A and (time) delays $d \in \mathbb{R}_{\geq 0}$. Hence a timed trace $\sigma \in (A \cup \mathbb{R}_{\geq 0})^*$ is of the form $\sigma = d_1 a_1 d_2 a_2 ... d_k a_k$ with $d_i \in R_{\geq 0}$ and $a_i \in A$, i.e., labels and delays are concatenated.

Definition 5.5 (rtioco)**.**

Let imp and s be $\mathcal{TIOTS}$:

$$imp \ \texttt{rtioco} \ s \leftrightarrow_{def} \forall \sigma \in Ttraces(e) : out(\langle imp, e \rangle \ after \ \sigma) \subseteq out(\langle s, e \rangle \ after \ \sigma) \quad (5.2)$$

where

- $\mathcal{TIOTS}$ is a timed I/O transitions systems. For details on $\mathcal{TIOTS}$, the reader is referred to the original publications of TRON [LMN04], since it is not necessary for the understanding of the application of TRON. It only should be noted that $\mathcal{TIOTS}$ have labels L that include distinct inputs L_i, outputs L_o as well as timed transitions ($d \in \mathbb{R}_{\geq 0}$) and an unobservable internal action label τ, i.e., $L = L_i \cup L_o \cup \{\tau\} \cup \mathbb{R}_{\geq 0}$.

- $\langle imp, e \rangle$ is the composition of implementation imp with the environment e. $\langle s, e \rangle$ denotes the composition of specification s with e.

- σ is an observable timed trace, i.e., $\sigma \in (L_i \cup L_o \cup \mathbb{R}_{\geq 0})^*$, i.e., a trace containing inputs, outputs and (time) delays.

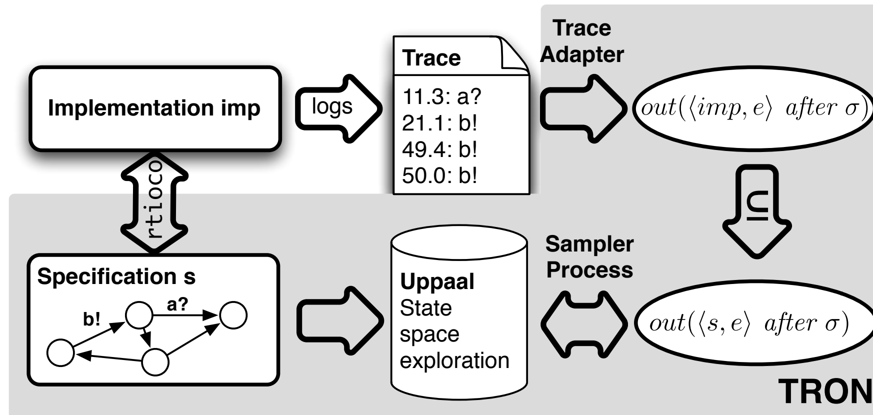

Figure 5.10: General overview for using TRON for PTT. TRON reads timed traces of the implementation *imp* via a trace adapter (cf. Sec 5.3.2). TRON uses the Uppaal Engine to compute the future state space of the specification s. Possible outputs are determined using a sampler process (cf. Sec 5.3.3). Actual outputs of the implementation are compared to outputs allowed by the specification as defined by the `rtioco` conformance relation.

- *after* σ denotes the set of possible states a system may be in after executing a timed trace σ.
- *Ttraces*(e) is the set of all possible timed traces of the environment e.
- *out*(K) is the set of outputs ($L_o \cup \mathbb{R}_{\geq 0}$) a $\mathcal{TIOTS}$ may produce from a set of states K.

Intuitively, `rtioco` denotes that when providing the same environment to specification and implementation and executing any timed trace from the environment, the outputs that one may see from the implementation model must be a subset of the specification model. Hence, the implementation has only behavior that is allowed by the specification. Delays and outputs are only allowed if they are specified in s.

Figure 5.10 shows a general overview for using TRON for PTT. TRON verifies that the outputs of the implementation $out(\langle i, e\rangle)$ *after* σ are included in the outputs allowed by the specification $out(\langle s, e\rangle)$ *after* σ. The discussion with respect to PTT only focuses on outputs of a system, i.e., there is no modeled input from the environment.

For determining the outputs of the implementation, TRON needs a connection to the specific implementation. For offline testing as considered in this work, a test adapter is provided that reads a textual trace. For accessing the timed automaton-based system description, TRON needs a sampler process to be provided by the test engineer. This sampler process uses dedicated signals and variables, allowing a comparison of input/output values of implementation and specification. In this work, the sampler process outputs

the system's power consumption at a given time, allowing its comparison to the respective value in the power trace at this instant. In order to understand the testing procedure, trace adapter and sampler process are briefly discussed in the following.

5.3.2 Trace adapter

TRON needs a connector to the implementation to read its outputs. For PTT, traces are already available from a given execution. For such offline testing, TRON provides a *trace adapter* that accepts a textual representation of the timed trace. Input actions, output actions and delays are defined. As previously mentioned, each action may have variables attached that are compared to the specification when the action is triggered, but not in any other case.

Listing 5.2 shows an exemplary trace used in the case study, where the physical quantity represents current draw under constant supply (i.e., power consumption). Lines 1 and 2 show the declaration of input and outputs actions and the associated (integer) variables: there are no input actions and only a single output action (`report`) with an associated variable for defining the power consumption (`crrValue`). Lines 3 and 4 are declarations for TRON parameters that include the `precision` of individual time units (here specified as $20\mu s$.), and a `timeout` that is used to signal when to stop the testing process[3]. The actual trace, which starts at line 6, is specified as a list of delays and output actions with a corresponding value. Delays are specified with respect to time units (`delay @6.0;`). Output actions are specified with the value of the associated variable at that moment in time (`output report1(4);`). The trace below describes that initially the current draw is $0\mu A$. After $120\mu s$, the current draw changes to $400\mu A$. Finally at $160\mu s$, the implementation draws $100\mu A$. For TRON such an execution trace for a given trace of measurements is generated based on the optimization described in Sec. 5.2.4.

5.3.3 Sampler process

TRON uses timed automata models of the specification and employs the Uppaal verification engine [MLN04] to compute the future state space of the specification (cf. lower right in Fig. 5.10). The *sampler process* is a model that specifies input and output actions of the specification. Figure 5.11 shows the *sampler process* used in the case study with a single output. The sampler process models the time when an output (`report`) may be generated. Additionally, it defines the range of possible values for the output variable `crrValue`. These

[3]The length of the test is determined in the offline case of PTT by the length of the power trace previously measured.

```
1  input ;
2  output report(crrValue);
3  precision 20;
4  timeout 20000000;
5
6  output report1(0);
7  delay @6.0;
8  output report1(400);
9  delay @8.0;
10 output report1(100);
11 ...
```

Listing 5.2: TRON trace format with a single output variable `crrValue` describing measured power consumption.

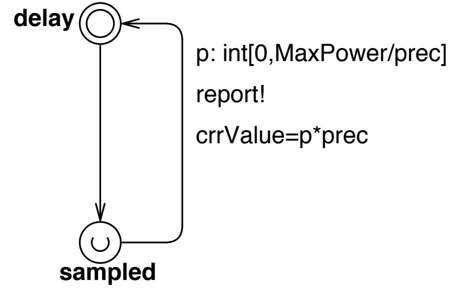

Figure 5.11: TRON sampler process for producing the output (`crrValue`) from the Uppaal model using a non-deterministic selection `p` with granularity `prec`.

are the same output action and variable as specified in Listing 5.2, i.e., trace and sampler process synchronize via `report` and `crrValue`.

On the transition, the current valuation of the power consumption variable `crrValue` is selected non-deterministically from the interval [0, `MaxPower`], which is achieved by using Uppaal's selections (here `p`). Selections non-deterministically assign an integer value within the given interval. To reduce granularity of possible values for `crrValue`, we first divide by a constant `prec` before the selection and then multiply by `prec` after the selection. Hence, `prec` defines the granularity of allowed power consumption values. An evaluation of `prec` is detailed in the case study (cf. Sec. 5.5.4.2). `MaxPower` is a constant, defining the maximum value for the power consumption of the complete system. Note that Uppaal's selections require constants in their ranges, i.e., the bounds of the interval need to be fixed at compile time. The sampler process shown in Fig. 5.11 does not describe any temporal behavior for the output; since power may be measured (and change) at any point in time, a change in power consumption is allowed at any time by the model. The sampler process is composed with the system specification model *Sys*. *Sys* is identical to the PTT-based approach.

5.3.4 PTT execution with TRON

For a better understanding, we may assume that a timed trace σ, as provided through the trace adapter, is executed. Let us further assume that from the last observation at time t_0 neither an input $i \in L_i$ nor an output $o \in L_o$ is provided by the implementation for δ time units ($\delta \in \mathbb{R}_{\geq 0}$). After this δ time units delay, the implementation produces a specific output $o \in L_o$. Remember, that the outputs of a $\mathcal{TIOTS}$ in states K are defined as $out(K)$ and that the set of outputs includes $L_o \cup \mathbb{R}_{\geq 0}$. Formally, this means that $\delta \in out(\langle i, e\rangle \textit{ after } \sigma)$ and $o \in out(\langle i, e\rangle \textit{ after } \sigma\delta)$, where $\sigma\delta$ denotes the concatenation of time delay δ to

the timed trace σ. For resolving this situation, TRON performs a state space exploration of the system model $\langle s, e\rangle$ and computes the largest delay d_{max} that is possible starting at t_0: $d_{max} = \max(d \in \mathbb{R}_{\geq 0} | d \in out(\langle s, e\rangle\ after\ \sigma))$. Given d_{max}, it can determined whether the delay δ is acceptable. If $d_{max} < \delta$ the delay is not acceptable and the test fails. In case $d_{max} \geq \delta$, the delay is allowed by the specification and TRON needs to check the respective target states for the possible output o. Once again this is done by state space exploration and the check whether $o \in out(\langle s, e\rangle\ after\ \sigma\delta)$ holds. If this is true the delay and output are appended to the timed trace: $\sigma' = \sigma\delta o$ and the exploration proceeds with all valid target states. This means that TRON needs to keep all potential target states in memory, such that it is capable of exploring all potentially valid traces. This yields that the reachable (valid) states are visited in a breadth-first search and stepwise manner. Contrary to this, the reachability check as performed by PTT can be organized in an arbitrary order, e.g., depth-first-search.

As pointed out above, TRON features the concept of a future size [KGL09] that is used to limit the pre-computation to δ_{max} time units. TRON features the command line option `-F` for specifying the future size δ_{max} (in time units). For PTT, this is important, since it determines the computational overhead of determining (future) outputs.[4] Consider the previous example of determining $\sigma' = \sigma\delta o$. Let us assume a delay transition of $\delta = 3000$. If $\delta_{max} = 1000$, TRON would need 3 separate pre-computation steps and arrive at a trace: $\sigma' = \sigma\delta_{max}\delta_{max}\delta_{max}o$. Choosing $\delta_{max} = 5000$ reduces the computational overhead by pre-computing the state space, since it uses just one exploration step: $\sigma' = \sigma\delta o$. The effect of the future size on checking power traces is explored in Sec. 5.5.4.1.

5.4 Case Study

In this case study, a Tmote Sky sensor node running the Harvester application (cf. Sec. 2.2) is monitored. A simplified version of the Harvester application is used that does not use the analog-to-digital converter to read sensor values but simply sends a given value periodically. Thus the components to be modeled are reduced to the two main contributors to power consumption: the microcontroller and the radio. Other components such as LEDs, sensors or the external flash are persistently powered off and are not included in the analysis. For the testcases, the power consumption of a sensor node is monitored by the voltage drop across a MHP201R0F 1Ω (±1% tolerance) shunt resistor measured by a digital multimeter (Agilent 34411A). Current draw is measured under a constant supply voltage. Current draw and power consumption are used interchangeably in the following. Note that for a variable supply voltage,

[4] The future size also plays a role in the testcase generation. This is however out-of-scope for PTT.

both voltage and current need to be measured in parallel to determine power consumption. A node is monitored for 20 sec. This is due to the limitation of 10^7 data points that can be stored by the multimeter. A plot of a time series of measurements is presented in Fig. 5.13, where the node periodically wakes up and turns on its radio in order to listen for a carrier. The first wake-up of Harvester occurs at 0.8s and the second one at $1.8s$. As can be seen from these wake-ups, Harvester is configured with a wake-up period of 1000 binary *ms* or approximately $0.977s$.[5]

5.4.1 Modeling the Harvester

Harvester is modeled as a network of timed automata. The Harvester is a complex system running an intricate software stack on top of a heterogeneous embedded system. The complexity of the system is managed through an abstract representation of expected behavior and component decomposition. In particular, individual models of Sys are differentiated between *hardware-oriented models* capturing the low-level behavior of individual hardware components and the corresponding low-level software (drivers), *software-oriented models*, representing the application-level software as well as the environment and testcase. Note that there is no claim on the soundness of abstraction and decomposition. As previously described, the system model is not a formal specification, but rather a specification of assumed behavior that is itself subject to refinements or improvements.

5.4.1.1 Modeling the hardware

The sensor node hardware is modeled as a network of timed automata, i.e., hardware components are modeled individually. Each system mode in a hardware component automaton has associated power bounds. Power consumption values for annotating the power bounds of the timed automata are based either on the data-sheet provided by the manufacturer or on characterization measurements. The system power consumption is the sum of the individual contributions as described in the invariant on system modes as shown in Listing 5.1.

For the analysis of the Harvester, the microcontroller and the radio need to be modeled focusing on their power modi in order to distinguish whether they are performing work, are idle or are powered off. Fig. 5.12 shows that the microcontroller has three modes: a low-power location (`OFF_and_LP`) that represents the processor turned off or in one of the MSP430 low-power modes (lpm1-4). In the `IDLE` location, the CPU is active and all clocks are enabled, but no computation is performed. The `ON` location denotes the power consumption of the microcontroller while performing computations. We refrain from timing

[5] One second contains 1024 binary milliseconds.

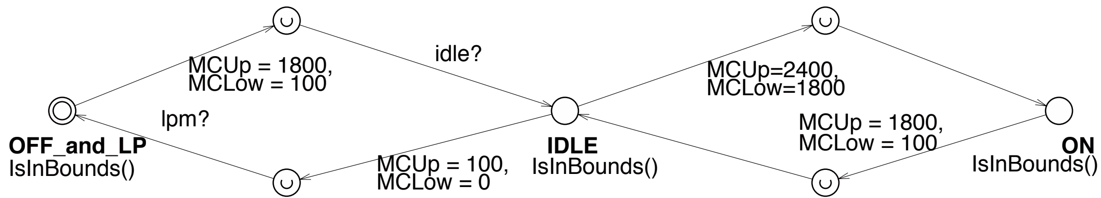

Figure 5.12: Model of the microcontroller hardware component. A corresponding software model controls the hardware component model using channels `idle` and `lpm`.

models for the microcontroller as transition times between the different modes are small.

Figure 5.5 shows the model based on the TI CC2420 radio datasheet[6]. On the upper left of the figure is the initial location that represents the radio being powered off. The upper half of the automaton models the startup of the radio until it reaches the default power-on location to the lower right (`LISTEN_AND_RECEIVE`). A corresponding magnified power trace of the startup behavior is depicted in Fig. 4.1.

The lower half describes the power consumption during the sending of a packet. The sequencing of steps in a single transmission uses a specification of the low-level radio driver of the radio to restrict possible state sequences. The naming of the locations is chosen accordingly: `AttemptSend` is the location where the clear channel assessment is performed. `ActualTransmit` is the location where the packet is transmitted, while `EFD` signifies a detected End Frame Delimiter, the end of the transmission.

5.4.1.2 Modeling the software

A software model controls a hardware model using binary channels. For exemplification one may refer to Fig. 5.5, where turning *on* and *off* the radio is achieved by binary channels. Sending is indicated with a channel label followed by an exclamation mark, e.g., the software turns on the radio via the symbol *on!* as seen in Fig. 5.6.

Due to the event-based, interrupt-driven nature of embedded systems, a representation of every possible program flow is difficult and error-prone [NS07]. Numerous possible interleavings are incurred by software running in a preemptable context. Moreover, differing semantics of the operating system scheduler considering hardware interrupts, software interrupts and deferred procedure calls as in TinyOS 2 make a comprehensive representation of a sensor node application impractical. Instead, the models employed focus on certain aspects of the system, e.g., the basic functionality of the MAC protocol, which considerably simplifies the software modeling. The MAC protocol is merely dependent on the radio. Its simplified operations, i.e., its use of the

[6] `http://focus.ti.com/docs/prod/folders/print/cc2420.html`

radio component, may be described in a declarative way: *The radio should listen to the channel about every second, if it is neither receiving nor sending. The on period should maximally last for 2.5 ms. In any other case, the radio should be turned off.* This behavior is modeled in the automaton illustrated in Fig. 5.6. It consists of five distinct locations `OFF`, `RadioOn`, `Send`, `sendDone` and `Receive` plus two transitional locations. When no communication is performed, the system toggles between `OFF` and `RadioOn`. This is achieved in Fig. 5.6 with the cycle containing the respective locations. As the specification describes, the software is expected to exit the `OFF` mode about every second, which is modeled as an invariant on the `OFF` location. Secondly, the invariant is added to have the radio in `RadioOn` for at most 2.5 ms. Note that this cycle within Fig. 5.6 includes a mode switch: Strict timing requirements are only enforced on consecutive wake-ups. Initially and on send operations, the requirements on being turned off for most of the period is disabled using the Boolean variable `initial` on exiting `OFF` location.

The remaining part for the given software model is dedicated to the radio's receiving or sending. This is achieved by equipping location `RadioOn` with two outgoing transitions `send?` and `receive?`. However, as enforced by Uppaal's synchronization semantics, these transitions can only be executed, if the signals `send!` and `receive!` are present. To do so one may specify some testcase automaton. Such a testcase automaton may non-deterministically emit the required signals or impose some restrictions on the timing.

The Harvester radio software model shown in Fig. 5.6 is simple due to its focus on a single functionality. Nevertheless it is a representative example of a software model in that it (1) defines the (de)activation of components (`on!`/`off!`), (2) defines a sequence of locations (e.g., `OFF- RadioOn- OFF- ...`), (3) features an additional, restrictive time bound on hardware locations (`RadioOn`) and (4) specifies intervals between locations.

5.4.1.3 Modeling test specifications

As pointed out before the usage of testcase automata for reducing the set of possible behaviors is advisory. For the 'Wake-up' testcase (cf. Fig. 5.13), an empty testcase automaton is used. This testcase automaton yields that the LPL MAC protocol as specified in Fig. 5.6 is restricted to the locations `IDLE` and `RadioOn`. This is because the software model never receives a signal to synchronize with from the environment. This basically models that the radio silently waits for incoming traffic. For analyzing a complex scenario (cf. Fig. 5.14), where packets are sent and received non-deterministically, a testcase automaton for emitting the respective `send!` and `receive!` signals is added.

For the sake of simplicity, the hardware and software models shown exhibit a single initial location such as the `OFF` location for the radio software model (cf. Fig. 5.6). In general, PTT may start at any system location and the system models need to accommodate for this fact. A single initial location is entered.

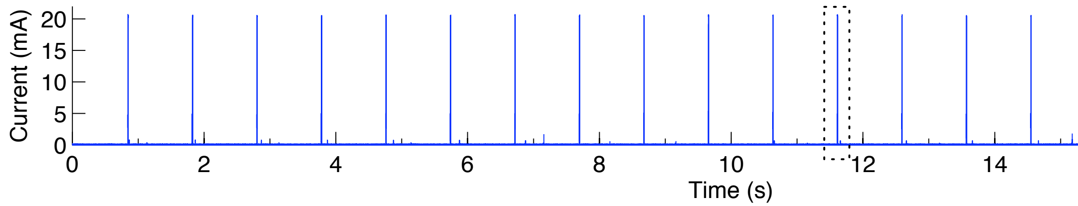

Figure 5.13: 'Wake-up' Trace: Periodic wake-up of the radio for listening on the channel. Annotated is the injection of an error by removing the wake-up between 11.608 and 11.618 seconds.

From this location, the timed automaton can unconditionally traverse to any other location and set the required context along an initial transition, e.g., the power bounds for a hardware automaton or the required synchronization for the software models.

5.4.2 Testcases

For the case study five different testcases are selected as discussed below.

5.4.2.1 Radio software testing

First, the 'Wake-up' PT in Fig. 5.13 is tested. Apart from validating the PT, a new trace is created by manually injecting an error ('Inject'). A single wake-up in the trace is removed between 11.6 and 11.62 seconds. This is marked in the figure. This injection models differences in temporal behavior. Hence, the sleep interval is prolonged representing defects such as missed interrupts or not setting the timer correctly. Uppaal verifies that this trace is not conformant to its specification. Binary search can be used to determine the last valid state that can be reached as 11.63*s*, i.e., shortly after the missing wake-up. Hence, a test engineer can conclude that a temporal requirement of the specification is not met due to a missing wake-up.

PTT is not restricted to simple examples. The elaborate trace ('Complex Trace') shown in Fig. 5.14 is analyzed using a general software model as presented in Fig. 5.5. In this case, an additional testcase automaton is used that allows the software model to receive and send messages at any instant in time.

5.4.2.2 Testing microcontroller low-power states

One of the common errors in programming low-power embedded systems is to forget to transfer the hardware into a low-power state. Given a testcase where no substantial processing is performed ('MC state'), a declarative specification of microcontroller operation can be formulated: *The processor should be maximally turned on for 10ms at a time. Subsequently it should be in a low-power state for at*

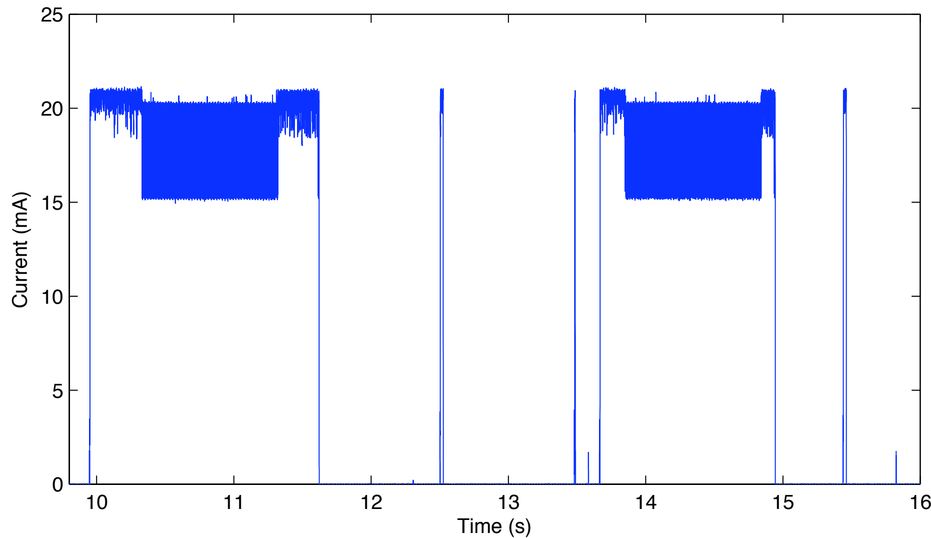

Figure 5.14: Power trace of a Tmote Sky sensor node. The sensor node performs send and receive operations.

least 2ms. Hence, it can be determined whether the microcontroller is operated correctly in a given PT. For validating this approach, an error is injected into the low-power scheduler by sometimes transferring to the regular idle mode instead of the lower power mode lpm4. In these specific sleep intervals, the system draws an additional current of about 1.8mA. This is detected by PTT, since a prolonged current draw can only be explained by an idle processor. However, this is prohibited by the specification.

5.4.2.3 Specification error

As a final testcase ('Specification'), an application of PTT is shown that detects an unexpected use of a hardware component that has not been modeled. This illustrates the capability of PTT to detect issues related to failures in the hardware or software modeling. In particular, the execution of a development version of Harvester is analyzed. During the sampling routine of the sensors, the development code turns on one of the LEDs for visual inspection. Figure 5.15 shows the current draw of the LED in such a case. Since the LED is not included in a hardware model, the reachability query fails.

The last reachable location can be determined with the time interval $[14.593s, 14.599s]$. This is exactly where the development code switched on the LED. Hence, hardware model issues can be identified, which can be revealed by an unexplainable power consumption pattern as in the case of the LED.

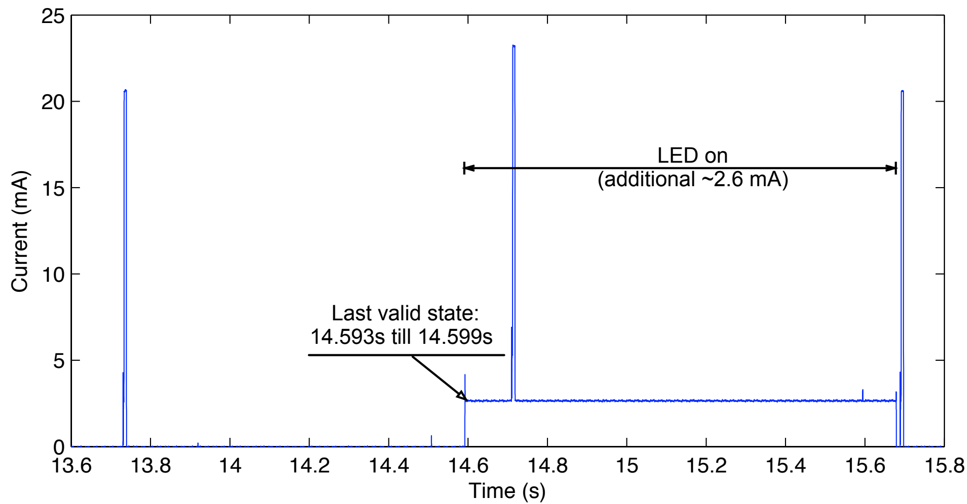

Figure 5.15: 'Specification' testcase: The LED is powered on, yet it is not included in the specification.

5.5 Empirical evaluation: Results and Benchmarks

In this section, experimental results are presented using a case study. The case study concerns the power consumption of a wireless sensor node. Firstly, the effect of the optimization presented in Sec. 5.2.4 on input problem sizes is described. Secondly, experimental results are presented for the implementations using TRON and Uppaal.

5.5.1 Power trace models

Table 5.1 presents a summary of the five testcases with respect to the original power trace sizes in number of individual measurements, and the effect of the quotient transition system optimization concerning the resulting number of locations in Uppaal.[7] As can be seen, for a typical example the optimization results in a compression of locations by at least an order of magnitude.

5.5.2 Experimental setup

All conformance tests were performed on a Sun-Fire-X2200-M2-64 blade running Linux. It features 2 dual core 64-bit AMD Opteron processors, i.e. 4 cores, running at 2.6 GHz. The blade has 8 GB RAM. The command-line verifier of Uppaal 4.1.2 is used (`verifyta`). In particular, `verifyta` is run with the `-u` option to obtain information about explored and stored states. For TRON version 1.4b5 is used, since tests with version 1.5 showed considerable degradations in

[7] This is obviously equivalent to the number of entries in the trace for TRON.

Model	Samples	Locations
Wake-up	1,000,000	1,141
Inject	990,000	1,087
Complex	310,000	23,418
MC state	1,000,000	1,293
Specification	1,000,000	1,336

Table 5.1: Measurement samples for each individual testcase and corresponding location count for the quotient transition system using the optimization described in Sec. 5.2.3.3

performance. TRON runs with a verbosity level of 8, in order to backup the state set and allow for final diagnostics for a failed conformance test. A logical (simulated or virtual) time clock (`-Q log`) is set. Unless otherwise noted, TRON uses a future size of 50,000 (`-F 50000`, cf. Sec. 5.3.4).

The main performance metric of these experiments is the execution time of the conformance test. The generation of the models from raw measurements is out-of-scope of this evaluation, but in the same order of magnitude for both tools. The models for Uppaal and TRON are the same for the system specification *Sys*; the obvious difference is that Uppaal includes the trace model *TM*, while TRON features the sampler process and an input trace as described in Sec. 5.3. For measuring time, the UNIX `time` facility is used and user times reported. Unless otherwise noted, the quotient transition system optimization is performed as described in Sec. 5.2.4.

5.5.3 Uppaal results

In the following different design decisions are evaluated based on i) the representation of time and ii) the representation of data values.

5.5.3.1 Measurement timing

The first experiments investigate whether the representation of time in the power trace makes a difference for Uppaal. For each power trace location, an invariant on the time a specific value was measured is used. In [WLT09] and Algorithm 5.1, the trace model *TM* in Uppaal uses relative durations for each measurement location. However, also an absolute time scale without resetting clocks after each power trace location can be employed. The difference is visualized in Fig. 5.16 and 5.17. As Table 5.2 shows, the runtimes of the relative approach are better. However, the size of the explored state space is comparable, indicating that Uppaal can internally better process short intervals rather than intervals with a large offset. Our experiments on all testcases show that using relative times performs generally better.

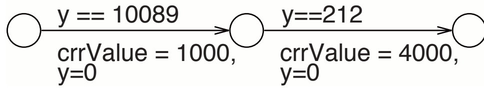

Figure 5.16: Relative time specification for power trace locations for clock y. Location invariants are omitted for the sake of a clearer presentation.

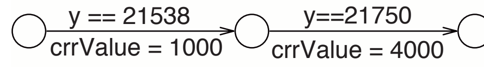

Figure 5.17: Absolute time specification for power trace locations for clock y. Location invariants are omitted for the sake of a clearer presentation.

Model	**States stored**		**States explored**		**Runtime**	
	rel.	abs.	rel.	abs.	rel.	abs.
Wake-up	78,394	77,058	139,456	152,591	3s	5s
Inject	42,605	41,889	63,697	74,087	2s	3s
Complex	2,059,630	2,065,845	2,083,177	2,099,871	816s	1102s
MC state	64,266	63,050	109,111	106,454	3s	5s
Specification	64,767	63,483	123,035	136,873	3s	5s

Table 5.2: Uppaal results: Comparison of absolute (abs.) and relative (rel.) time specification for power trace locations.

5.5.3.2 Measurement granularity

Similarly, Table 5.3 presents the results for the different testcases concerning different measurement granularities. Initially a resolution for 1 μA of measurements was used. As a second step, a restricted granularity of 100 μA was employed. This experiment tries to explore whether the size of the value domain has an effect on the size of the state space and runtime in Uppaal. As Table 5.3 shows, the results for the size of the state space are the same and runtime results are comparable.

Model	**States visited**		**States stored**		**Runtime**	
	1 μA	100 μA	1 μA	100 μA	1 μA	100 μA
Wake-up	78,394	78,394	139,456	139,456	3s	3s
Inject	42,605	42,605	63,697	63,697	2s	2s
Complex	2,059,630	2,059,630	2,083,177	2,083,177	827s	755s
MC state	64,266	64,266	109,111	109,111	3s	3s
Specification	64,767	64,767	123,035	123,035	3s	3s

Table 5.3: Uppaal results: Comparison of different granularities of power trace measurement values $\{1\mu A, 100\mu A\}$.

Model	Future size					
	$1,000$	$10,000$	$50,000$	$100,000$	$200,000$	$500,000$
Wake-up	$535s$	$468s$	$456s$	$461s$	$462s$	$463s$
Inject	$242s$	$220s$	$214s$	$214s$	$213s$	$214s$
Complex	$25,456s$	$26,673s$	$26,482s$	$26,682s$	$26,385s$	$26,273s$
MC state	$255s$	$241s$	$240s$	$238s$	$243s$	$244s$
Specification	$236s$	$221s$	$223s$	$225s$	$225s$	$222s$

Table 5.4: TRON results: Comparison of different future sizes using a 1 μA granularity in the sampler process.

5.5.4 TRON results

In a similar vein, the performance in TRON for measurement granularity and representation of time was explored.

5.5.4.1 Measurement timing

For representation of time, the effect of different future sizes was explored. The future option in TRON (`-F`) describes the number of time units the state space of the specification is pre-computed. If this future window is short, TRON has to go through multiple state space explorations as explained in Sec. 5.3.4. If the future size parameter is too large, too much of the future state space is explored; an output may be previously seen from the implementation. Note that the Harvester model is periodic with 50,000 time units due to its wake-up behavior, i.e., there is always an output within an interval less than this period (cf. Fig. 5.13). Table 5.4 shows the exploration for different future sizes. The results show all future sizes perform equally as long as the state space pre-computation is larger than the periodicity of the software, i.e., $\geq 50,000$ time unites. This is because larger future size values do not result in an expensive exploration: As the models have a periodic behavior, the state space exploration is finished after one wake-up period. Hence, increasing the future size value larger than the period does not result in any exploration overhead. This is indicated by the results shown in Table 5.4. However, there is a penalty for shorter values, since the state space exploration is cut into smaller pieces. This generates an overhead in terms of individual iterations of explorations (cf. Sec. 5.3.4).

5.5.4.2 Measurement granularity

Table 5.5 presents the results for TRON concerning measurement granularity ($1\mu A$ and $100\mu A$) for the different testcases. Note that in this case, the value domain makes a significant difference. The difference stems from the coupling through the sampler process: TRON needs to compute the future state space

Model	Granularity		
	$1\ \mu A$	$100\ \mu A$	GCI-based
Wake-up	$456s$	$254s$	$210s$
Inject	$214s$	$120s$	$92s$
Complex	$26,482s$	$9,620s$	$13,263s$
MC state	$240s$	$131s$	$98s$
Specification	$223s$	$122s$	$93s$

Table 5.5: TRON results: Comparison of different measurement granularities for the sampler process using a future size of 50,000.

of the specification and then performs a comparison with the implementation trace. The sampler process allows the specification to select any integer value within the given bounds. Hence, the finer the granularity, the larger the sets to be compared. The interface between the Uppaal exploration and TRON needs to compare each possible output with the trace output of the implementation. This comparison creates a considerable overhead.

In order to allow for the largest minimization of the number of measurement values, a trace abstraction based on GCIs was performed: Each measured power consumption value is annotated only with its corresponding GCI in the trace (cf. Sec. 5.2.3.3). In turn, the specification needs to be extended to associate GCIs with system states. To this end, the invariant function of the specification model is changed as shown in Listing 5.3. The number of possible values in the sampler can be reduced from 262 (for 100 μA granularity) down to 42 (GCI) values. This allows for some further improvements as shown in the right column of Table 5.5. 'Complex' apparently does not benefit from a reduced representation in the sampler process, probably due to the large degree of non-determinism. It seems that computing the value of `actual` for each location invariant also creates some overhead.

```
bool IsInBounds(){
   power_t actual = (intervals[crrValue]+intervals[crrValue+1])/2;
      if(actual > RadioUp+MCUp or actual < RadioLow+MCLow) return(false);
      return(true);

}
```

Listing 5.3: Power consumption invariant for GCI-based description of trace. `crrValue` denotes the GCI associated with the measurement. The corresponding power consumption `actual` is computed online in the invariant function.

5.5.5 Uppaal versus TRON comparison

Lastly, the runtime of the best TRON version with the best Uppaal version is compared to give an overall impression of their relative performance. The

Model	Uppaal	TRON
Wake-up	$3s$	$210s$
Inject	$2s$	$92s$
Complex	$755s$	$9,620s$
MC state	$3s$	$98s$
Specification	$3s$	$93s$

Table 5.6: Comparison of the best runtimes for Uppaal and for TRON

results are shown in Table 5.6. Uppaal outperforms TRON by at least an order of magnitude.

While initially better performance using TRON was expected, its performance can be explained by its typical usage as an *online* testing tool:

1. In offline testing, one exactly knows the time of the next event and can in turn do a limited exploration for exactly this duration. Additionally, in offline testing one knows exactly what the next state must be and performs an exploration only for this particular future state. In contrast, TRON has no knowledge of future states and must perform a (more expensive) complete exploration. In a second step it must perform a comparison between the (large) set of explored, possible states and match them with the set of states allowed by the trace. This happens at each synchronization point, i.e., when there is an output in the timed trace of the implementation.

2. TRON naturally finds the last, "deadly transition" (cf. Sec. 5.2.2.3). Hence, there is no need to perform a search as in the Uppaal-based version. Uppaal approximately needs $log_2(n)$ runs of the model checker to determine the failing location, where n is the number of locations in *TM*. This results in about 11-15 runs for the traces used in the case study.

5.6 Summary

PTT is a novel method to test a WSN using power measurements. The problem can be mapped into the class of timed automata by exploiting discrete value ranges of the continuous power measurements. This chapter presents optimizations that allow PTT to use standard timed verification tools for conformance testing of WSN based on power measurements. In particular, it presents implementation based on a timed model-checker, Uppaal [BY04, BDL04], and a timed online-testing tool, TRON [LMN04]. A comparison of the two approaches shows that the Uppaal-based implementation outperforms the TRON-based implementation.

5.6.1 Related Work

PTT is the first model-based approach to exploit power measurements of a WSN to test for conformance to a specification. It is mainly inspired by the previous work on testing power consumption of a system using actual hardware described in the previous chapter. Typically physical quantities, such as power and energy consumption, are rather explored in simulation than in real tests, e.g., for sensor networks in [SHC+04, LWG05]. Simulation relies on the accuracy and fidelity of models of power consumption. In comparison, testing power consumption using actual measurements assures that such models actually hold for real devices.

PTT is based on conformance testing. There is related work concerning conformance relations both untimed [Tre94] and timed [LMN04] and even for hybrid system descriptions [vO06]. Most closely PTT is related to the work by Bohnenkamp et al. on quantitative testing [BS08a]. The main differences is that in quantitative testing the uncertainty of the measurement is constant throughout the specification, i.e., there is a behavior at a distance of at most $x, x \in \mathbb{R}_{\geq 0}$. However, hardware component states may have differing uncertainties depending on the system mode. Hence, a mode-based uncertainty as employed in PTT by using intervals on individual system modes of hardware components is preferable. Moreover, for quantitative testing there is currently no tool support. The presented evaluation includes TRON [LMN04], since it is an available, maintained tool and allows a direct comparison with an Uppaal-based approach. The use of hybrid systems for power consumption is described in detail in the discussion below.

The conformance test between measurements and specification relies on timed trace inclusion, i.e., whether a timed trace, the measurements, is included in a timed automaton model, the specification. However, timed trace inclusion as discussed in the context of generation of a validation automaton [BS08b] or in the context of abstraction refinements [JLS00] is complementary to this work, since PTT is focused on using available tools for conformance testing.

5.6.2 Discussion

Before discussing other approaches to conformance testing of power measurements, there is one difference worth mentioning between PTT and the power unit tests described in the previous chapter. While both use bounds on power consumption, PTT does not specifically define or utilize an uncertainty on transition times between system modes. Note that uncertainty with respect to timing can still be included in the models by the specification engineer as shown in Fig. 5.5 and Fig. 5.6: For the start-up behavior of the radio there are only approximate upper bounds on the timing behavior in this case. However, the method does not exploit this uncertainty, since, different to power consump-

tion, clock variables are not discrete (but real) variables and the optimizations as performed for the power consumption do not apply here.

5.6.2.1 Other automata-centric options

Obvious choices for trace verification are *hybrid model checkers* such as HyTech [HHWT97] or PHaver [Fre08]. The power trace itself can be better described by a hybrid automaton that allows models to use continuous variables. For the power trace measurements a representation as a continuous variable is suitable. However, the system specification abstracts away from the continuous properties of measurements by using bounds. Additionally, the transitional characteristics of power consumption are typically not of interest. Rather the power consumption in a given system mode needs to be checked. The formulation as a hybrid automaton does not provide any benefit in modeling when using bounds on system modes. In contrast, model checking of hybrid automata is a more difficult problem than model checking of timed automata. In conclusion, PTT does not benefit from employing hybrid model checkers.

Another option is the discretization of time. Fundamentally, since we deal with a microprocessor that is synchronous with its clock cycle (for sensor networks in the order of tens of μs), we can safely abstract away from continuous time to discrete time. In turn, the problem is discretized and model checkers for untimed automata may be used such as SPIN[8] or NuSMV[9]. Compared to the blow-up introduced by discretization of time, the representation of time in Uppaal using symbolic methods seems rather efficient and is not perceived as the major issue of the limited performance as discussed in the following.

5.6.2.2 The issue with model checking

The fundamental problem of a model checker based approach to PTT is that the model checker cannot exploit the simple, linear structure of the trace. Model checkers must store visited states and hence typically run into memory problems because of state-space explosion. This issue is further elaborated on in the following chapter, since it is even more profound when concurrently testing multiple, communicating sensor nodes.

[8] `http://spinroot.com/`

[9] `http://nusmv.irst.itc.it`

6

PTT for communicating sensor nodes

PTT is a method for determining whether a (finite) timed trace of power measurements is included in the traces of a specification of a WSN. However, as discussed before, PTT is hampered by the so-called state-space explosion problem: The expansion of all possible system behaviors may yield a large number of system states, which is exponential in the number of a system's concurrently executed activities. PTT faces a high degree of non-determinism, since it needs to jointly execute timed automata-based models of multiple, communicating sensor nodes. Sensor nodes operate autonomously and only communicate sporadically; they show a high degree of concurrency. This may result in a very large number of executions that need to be checked against the observed power traces. A good state space exploration strategy would be needed that produces the shortest witness that shows that the system model agrees with the power measurements; yet it is unclear how such a strategy would operate. Hence, PTT can be costly, particularly in terms of memory consumption. The goal of this chapter is to scale PTT to formally reason about the correctness of multiple, communicating sensor nodes. To this end, this chapter provides the following contributions:

- It identifies the model checker as the memory bottleneck for PTT.
- It describes two domain-specific properties that can be used for reducing memory consumption of PTT.
- It describes a sound and complete test procedure exploiting these domain-specific properties.
- It shows in a case study that the novel test procedure allows an engineer to test concurrent power measurements of two communicating sensor nodes.

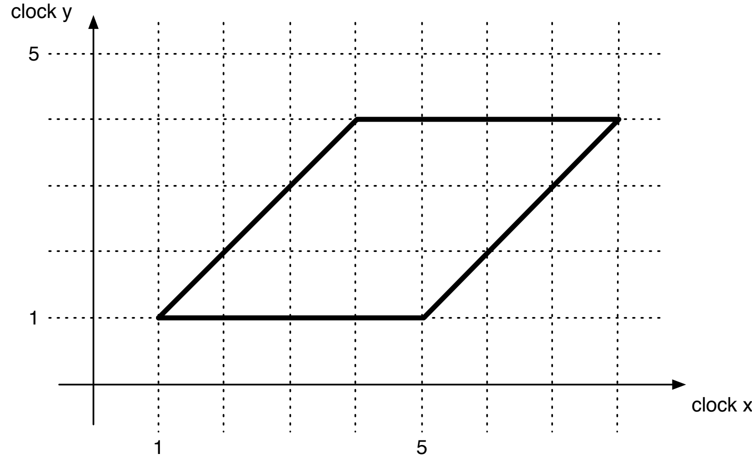

Figure 6.1: A clock zone for two clocks x and y that is defined by $1 \leq x \leq 8 \wedge 1 \leq y \leq 4 \wedge x - y \geq 0 \wedge x - y \leq 4$.

6.1 Background Theory

For the understanding of this chapter, some more background with respect to the reachability analysis of a model checker for timed automata is necessary. The analysis of reachable sets of states for timed automata exploits the partitioning of clock valuations into distinct sets of assignments. *Regions* are equivalence classes of an equivalence relation over clock valuations [Yov98, AD94]. Exploiting regions, a corresponding *region automaton* can be constructed from the original timed automaton. The region automaton is a finite state model. The set of regions is finite. However the number of regions is exponential in the number of clocks [Yov98]. To this end, a more efficient representation in the form of *zones* is used in model checkers such as Uppaal. A zone is the solution set of a clock constraint that is maximal with respect to the clock assignments satisfying the constraint. This provides a coarser and often more compact representation. As an example consider Fig. 6.1: it features a single zone that is defined by $1 \leq x \leq 8 \wedge 1 \leq y \leq 4 \wedge x - y \geq 0 \wedge x - y \leq 4$, yet numerous regions (e.g., one region for each of the 28 line segments of length 1).

We are interested in the reachability graph of a timed automaton. Let the system state s of a reachability graph be a triple:

$$s = (\vec{l}, U, V),$$

where:

- $\vec{l} = (l_0, ..., l_n)$ is a location vector. l_i corresponds to the current location of the i'th component automaton of a network of timed automata.
- U is a set of clock constraints defining the set of allowed clock valuations in the current locations.
- V is the set of variable valuations.

AP is the set of atomic propositions of a timed automaton. $L(s) \rightarrow 2^{AP}$ denotes the atomic propositions associated with a state s, i.e., the labels associated with the corresponding location vector $\vec{l}$.

A path in the reachability graph is a sequence of states: $\pi(s) = s, ..., s', s'', ...$ where s is the initial state and each pair of adjunct states s', s'' corresponds to a state-to-state transition in the reachability graph resulting from delay and discrete transition of a timed automaton. The number of system states is infinite, but the possible states can be grouped into a finite set of equivalence classes using regions or zones as described above. The reachability query, used in PTT, $E <> final$ searches for a path $\pi^f = s, ..., s_f$, where $final \in L(s_f)$. $final$ is the label of the final location of TM, i.e., $final \in L(l^f_{TM})$.

6.2 Scalability, an open issue

6.2.1 Problem

The fundamental problem of a model checker-based approach is state-space explosion: the model checker cannot exploit the simple, linear structure of the trace model TM. In general, model checkers need to store all already visited states allowing to decide whether a state is newly reached and needs further exploration, or has been visited before.

Experiments were performed to investigate scalability of Uppaal with respect to trace length. To this end, a looping over TM was used to increase the length of the power trace model and investigate the effect on Uppaal's memory consumption. Two different power trace models (TM) were used:

- `Idle` is an artificial power trace model where the system never leaves a low-power mode.
- `Wake − up` is the testcase described in Sec. 5.4.2.

Let Sys be the system model as described in Sec. 5.4.1, i.e., a system model of a single sensor node. For benchmarking PTT, the following properties were verified: (a) $\texttt{Idle} \models E <> final$, (b) $\texttt{Wake} - \texttt{up} \models E <> final$, (c) $\texttt{Idle}||Sys \models E <> final$ and (d) $\texttt{Wake} - \texttt{up}||Sys \models E <> final$.

The obtained results are shown in Fig. 6.2: The number of reachable states is shown in relation to the number of locations of the power trace model TM, i.e., here for `Idle` and `Wake − up`. The figure shows that the number of stored states is proportional to the number of locations of the corresponding power trace model. When executing the reachability analysis the accumulation of states consumes significant memory. Fig. 6.3 shows the ratio of the number of states of a composed model ($TM||Sys$) to the number of states of a trace-only model (TM) is constant. Hence, the composition of TM with the system model

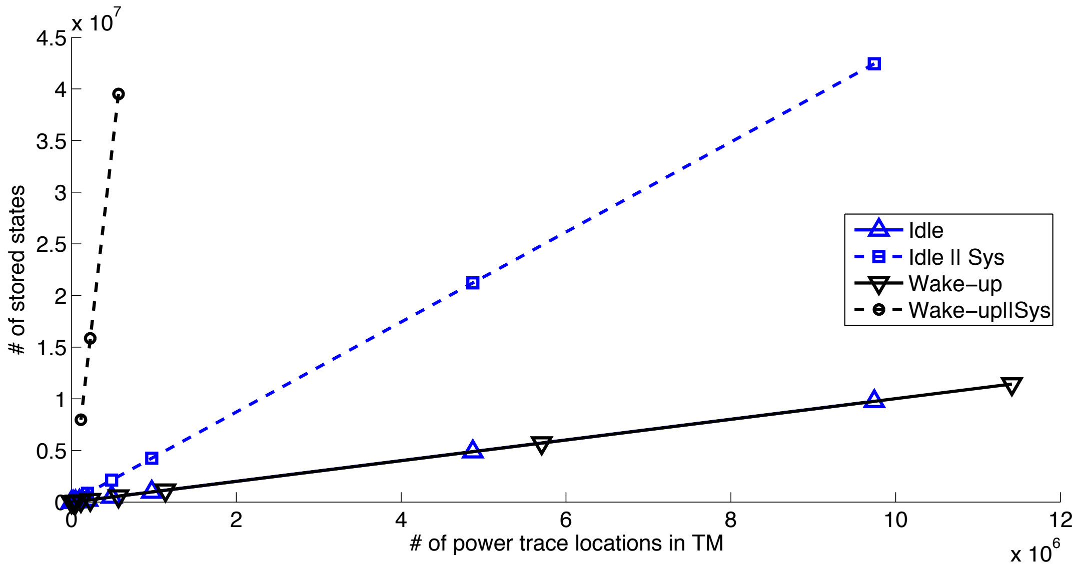

Figure 6.2: Number of stored states versus number of locations of the trace model. Experiments are performed on *TM* alone as well as on the composed model *TM*||*Sys*.

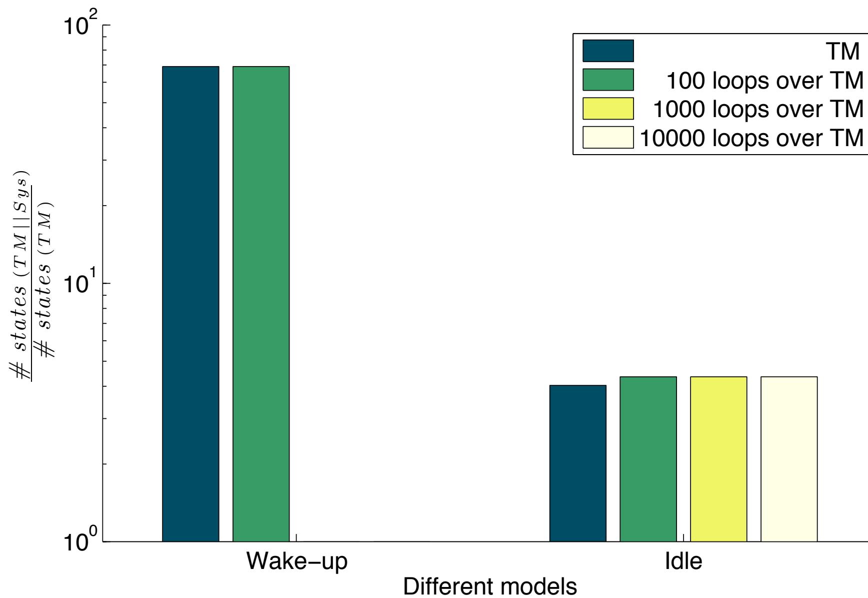

Figure 6.3: Ratio of the number of states of a composed model (*TM*||*Sys*) and trace only (*TM*), i.e., $\frac{\#\,states(TM||Sys)}{\#\,states(TM)}$, for different numbers of looping rounds. (Longer runs for 'Wake-up' did not finish.)

Sys generates a proportionally constant overhead for the state space. Hence, a major impediment for effective PTT is that the state space grows linearly with the number of locations in the power trace model. Fig. 6.2 also shows that this problem is exacerbated when there is more activity in the power

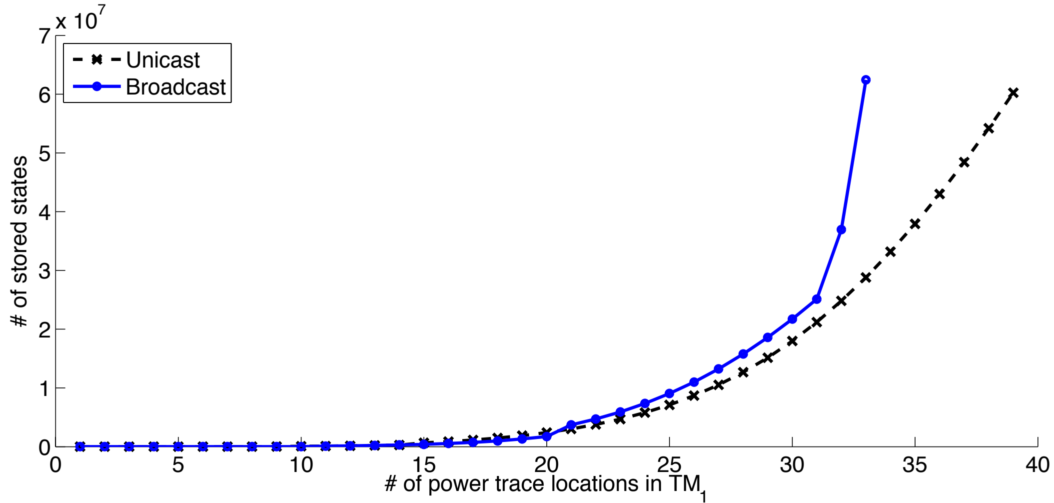

Figure 6.4: Number of states stored during the state space exploration of $Sys_1||TM_1||Sys_2||TM_2 \models E <> (final_1 \wedge final_2)$ versus the number of power trace locations in TM_1 for two different testcases.

trace: `Idle`$||Sys$ does not expose the non-determinism inherent in the system model. In comparison, the steep slope of `Wake − up`$||Sys$ indicates the increased non-determinism resulting from a power trace with more activity.

This problem of scaling PTT worsens for system models containing multiple sensor nodes. To this end, a simple experiment was performed to show the complexity of concurrently checking synchronously measured power traces of communicating sensor nodes (cf. Sec. 6.3.1 for details). It was tested whether two traces are conformant to the concurrently executed system models, i.e., whether $Sys_1||TM_1||Sys_2||TM_2 \models E <> (final_1 \wedge final_2)$ holds. Sys_i is the model of sensor node i (component models for hardware and software parts), TM_i is the model of the power measurements of sensor node i. $final_i$ is the label of the final location of power trace model TM_i. The system models Sys_i of the sensor nodes include models of the communication between nodes. Hence, Sys_1 and Sys_2 may synchronize via these communication models. For simplicity, we iteratively checked for the reachability of a dedicated location in one of the power trace models, TM_1, i.e., the trace model for the first sensor node. The result is shown in Fig. 6.4. After reaching the 30th location of TM_1, Uppaal has already stored $17,986,721$ states. This prohibits timed verification of more complex models as memory of commodity computers is limited.

Both experiments illustrate that timed verification is severely hampered by state-space explosion. This also applies to PTT even though it only focuses on the inclusion of finite timed traces in the behavior of a timed automata-based system model. Testing the conformance of communicating sensor nodes is the main motivation for this chapter. For ease of explanation, we focus on two

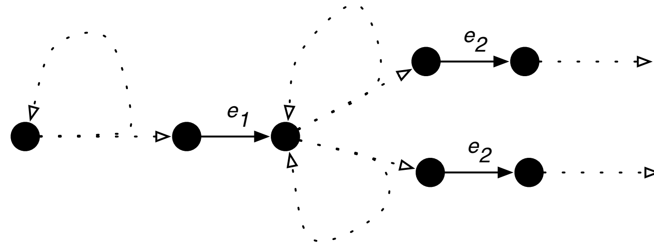

Figure 6.5: There may be loops in the reachability graph of $TM||Sys$. However, there are no loops across multiple locations in TM. Once TM traverses to the next location via a discrete transition e_i, all previous visited states may be discarded.

communicating sensor nodes in the following. The extension to more sensor nodes is straight-forward.

6.2.2 Property of the power trace model: No absorbing loops.

The power trace model TM can be exploited by the following fact: TM is a timed automaton with a single line of development, no alternative system evolutions are possible. We refer to this as *linear behavior*. When being composed with the system model Sys, the resulting transition system of $TM||Sys$ does not contain loops that can be visited infinitely often as time progresses. This is because the power measurement captured by TM evolves over time. This linear behavior can be exploited for discarding all previously visited states in the state space exploration, when TM traverses to the next location. This is illustrated in Fig. 6.5, where discrete transitions between locations in TM are denoted as e_i. There is no access to the internals of Uppaal, yet there is a need to exploit its elaborate mechanisms in terms of state space exploration.

The ultimate goal of this chapter is to test for conformance of synchronously measured power traces of two communicating sensor nodes. Hence, we need to test whether $Sys_1||TM_1||Sys_2||TM_2 \models E <> (final_1 \land final_2)$ holds. The conformance test procedure needs to exploit the linear behavior of the trace models TM_1, TM_2 to mitigate state-space explosion. The main idea is to segment trace models TM_1, TM_2 into m smaller pieces $TM_1^j, TM_2^j, j \in \{1, 2, ..., m\}$. Let π^f be a path that leads from the initial state to a state s_f, with $final \in L(s_f)$ as defined above. Instead of generating the complete path, the proposed approach produces subsegments of the path, such that $\pi^f = \pi_1 \circ \pi_2 \circ ... \circ \pi_n$. Note that $\pi_j \circ \pi_{j+1}$ denotes the concatenation of path π_j , which is generated for checking the conformance of trace segments TM_1^j, TM_2^j, and path π_{j+1}, which is generated for checking the conformance of trace segments TM_1^{j+1}, TM_2^{j+1}. Segmentation can be performed at any point. Using this segmentation, two trace models TM_1, TM_2 of concurrently measured power consumption are defined as PTT-conformant with respect to the system models Sys_1 and Sys_2 if the following

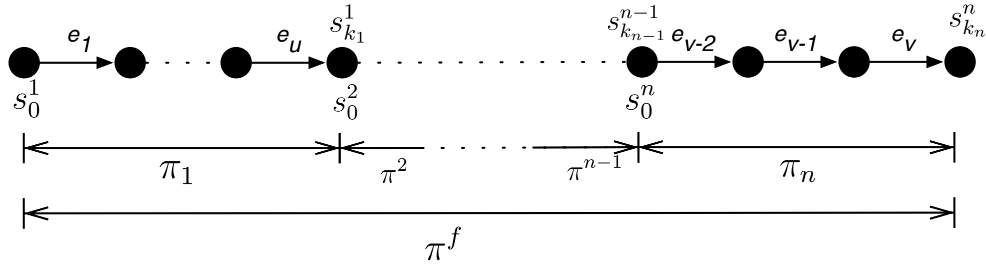

Figure 6.6: A path in the model is segmented into segments: $\pi^f = \pi_1 \circ \pi_2 \circ ... \circ \pi_n$. e_i denote discrete transitions in *TM*.

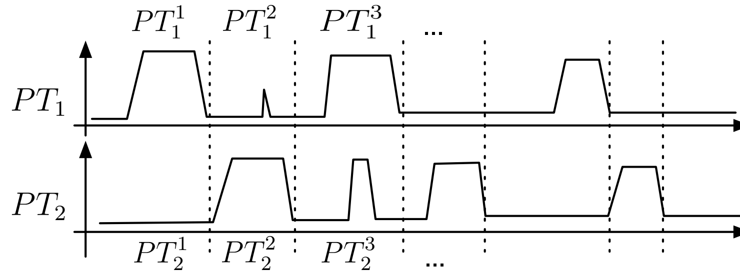

Figure 6.7: The main idea for scalability: Power traces PT_1 and PT_2 can be cut into segments PT^j_1 and PT^j_2, $j \in \{1, 2, ..., m\}$.

holds:

$$\forall j \in 1, .., m : Sys_1 || TM^j_1 || Sys_2 || TM^j_2 || applyContext(con^{j-1}) \models E <> final^j_1 \wedge final^j_2,$$

where TM^j_1, TM^j_2 are segments of the trace model. Since clocks progress synchronously in a network of timed automata, such segments model the same time interval for synchronous measurements. $final^j_1, final^j_2$ are labels of the final locations of these segments. *applyContext*(*con*) is a model for applying an initial state *con* to a segment as described below. This model is needed as the segmentation requires to restore the *context* (state) con^{j-1} of a path when verifying the succeeding segment *j*. This is because $\pi_1 \circ \pi_2 \circ ... \circ \pi_n$ needs to be equivalent to π^f as illustrated in Fig. 6.6 and described below. Hence, *applyContext* takes the context of the previous segment con^{j-1} and applies it as the initial state to the current segment *j*. Note that con^0 is the initial state of the system models $Sys_1 || Sys_2$.

Let us denote PT_1, PT_2 as the actual measured traces of power consumption. A visualization of the segmentation based on PT_1 and PT_2 is shown in Fig. 6.7. This segmentation strategy tries to keep the memory consumption of Uppaal for the individual segments below a threshold. This tries to render the conformance test of two communicating nodes with respect to the synchronously obtained power measurements feasible.

6.2.2.1 Context restoration

As pointed out above, the aim is to generate a path π^f that verifies the conformance, i.e., $Sys_1||TM_1||Sys_2||TM_2 \models E <> (final_1 \land final_2)$. Hence, the last state generated when testing segment j must match the initial state when testing the segment $j+1$, so that $\pi^f = \pi_1(s_0^1) \circ \pi_2(s_0^2) \circ \pi_3(s_0^3) \circ ... \circ \pi_m(s_0^m)$, where $\pi_1(s_0^1) = s_0^1, s_1^1, ..., s_{k_1}^1; \pi_2(s_0^2) = s_0^2, s_1^2, ..., s_{k_2}^2; ...; \pi_m(s_0^m) = s_0^m, s_1^m, ..., s_{k_m}^m$ and $\forall i \in \{2, ..., m\} : s_0^i = s_{k_{i-1}}^{i-1}$.

A *context* is a state consisting of a location vector, a set of variable constraints for discrete variables, and a set of clock constraints describing the zone of valid clock valuations. The context of segment j is produced by generating a counterexample with the model checker using the inverted query: $A[]\ \texttt{not}(final_1^j \land final_2^j)$. Note that $A[]$ means *always invariantly*, i.e., this property must always hold. Note that there may be several valid contexts for a segment given by different paths. The model checker provides a single context for a segment of a specific path in the form of a counterexample. If multiple contexts need to be determined, a previous context *prev* can be excluded using the reachability query, e.g., for Uppaal: $A[]\ \texttt{not}(final_1^j \land final_2^j \land \texttt{not}\ prev)$. An example is the black circle shown in Fig. 6.9; the corresponding query in Uppaal is: `A[] not (final_1 and final_2 and not(x==3 and y==2))`

6.2.2.2 Applying context

The context provided by the model checker is applied as the initialization (state) for the consecutive segment. This initialization includes the setting of the initial location of each timed automaton and of the discrete variables and the clocks. A dedicated initialization model, *applyContext*(*con*), is used for the initialization of the context *con*. Fig. 6.8 outlines a conceptual model for setting the clocks using an initialization model *applyContext*.

The initialization of clocks requires to apply a set of clock assignments corresponding to the zone (i.e., clock constraints) of the context. This application of a complete zone may result in a large model. This is because the initialization model needs to non-deterministically sample from the complete set of allowed clock assignments. As a remedy to a large model due to the initialization using zones, we can also sample from a given zone and initialize the consecutive segment with a *discrete (integral) context*. Fig. 6.9 depicts the approach for two clocks. In the figure, discrete valuations (circles) for each clock are selected from a given zone. Only a single discrete context is selected at a time, e.g., the black circle.

Note that the testing method remains sound and complete, since the specification model *Sys* and the power trace model *TM* belong to the class of closed timed automata. Closed timed automata only include (positive boolean combinations of) constraints on clocks in the form of $x \leq c$ and $x \geq c$, where x is

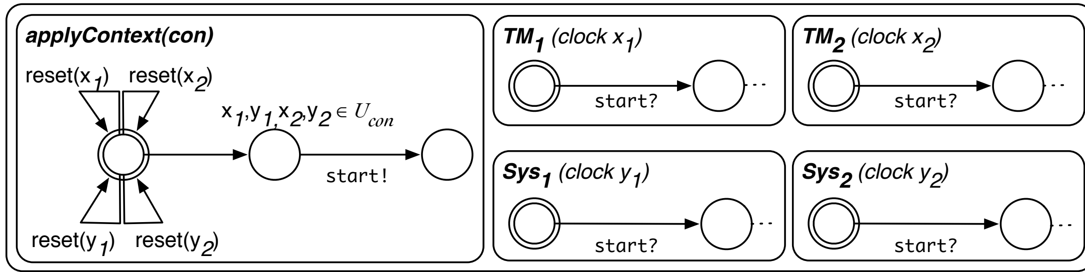

Figure 6.8: Conceptual model for initializing context in Uppaal using an initialization model *applyContext*. Since Uppaal does not allow models to set initial values of clocks, clocks need to be increased in an initialization phase before starting the actual exploration. *applyContext* sets the clocks x_1 of TM_1, x_2 of TM_2, y_1 of Sys_1, and y_2 of Sys_2 corresponding to the constraints of the context $con = (\vec{l}_{con}, U_{con}, V_{con})$. When the clocks are set to satisfy the constraints $(x_1, x_2, y_1, y_2 \in U_{con})$, the actual exploration is started by broadcasting a `start` signal.

a clock and c is an integer constant. Closed timed automata are closed under digitization. The reachability query is a qualitative property, closed under inverse digitization. Hence, an integral-time model can be used when transferring context between segments [HMP92, AM04]. Note however that choosing a single, discrete (integral) context means in the worst case an large number of contexts need to be consecutively applied for each segment. However, the following case study shows that this approach works quite well in practice as the specification provides some leeway in terms of timing requirements.

6.2.3 Property of low-power (embedded) systems: Recurrent identifiable locations.

There is a second property, which is inherent to low-power systems, that can be used to select when a trace *PT* should be segmented. For low-power embedded systems, there is a dedicated recurrent location in each component model. This is the low-power sleep mode, where the system does not need to perform any task and switches to a low-power mode. The power consumption in the low-power mode is unique and can be unambiguously associated with a measured value in the power trace. In case of a sensor node, this is exemplified by the radio component (cf. Fig. 6.11 and Fig. 6.12), which is the major contributor to power consumption. After each operation, whether sending or receiving, the radio needs to return to a low-power mode. In between these sleep modes several operations can be performed that have different effects on the system state, e.g., by setting variables or resetting clocks.

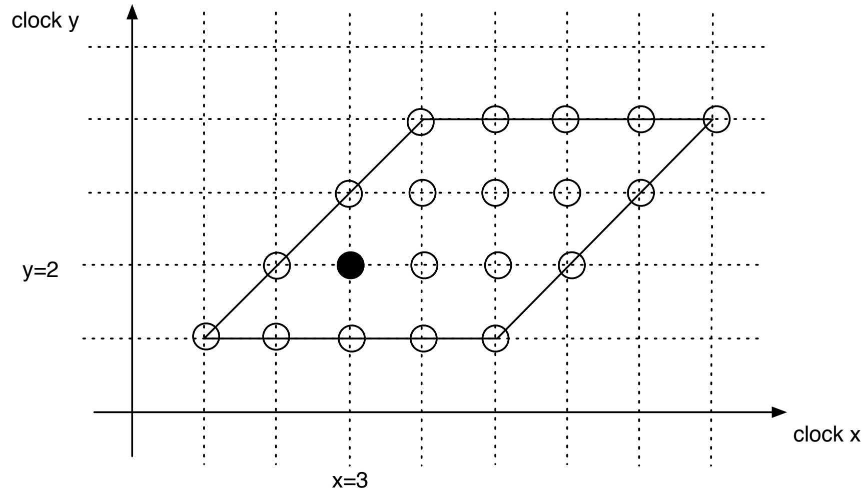

Figure 6.9: Integral contexts for clocks x and y are selected from a zone for two clocks. The black circle shows one selected context with $x = 3 \wedge y = 2$.

6.2.3.1 Identifying recurrent locations

Figure 6.10 shows that for a simplified system model, there is a location l_{uniq} the timed automaton always returns to. In a network of timed automata, this is obviously a location vector $\vec{l}_{uniq}$, i.e., a unique location for each of the individual timed automata. Now, we can select $\vec{l}_{uniq}$ and also mark the corresponding measurement locations in TM, as this recurrent location can be associated with a unique power measurement.

6.2.3.2 Segmentation of traces

Traces may be segmented at any point. Such points are denoted *segmentation points* in the following. In this work, segmentation points are selected such that each trace is marked with the uniquely identifiable location of the system model as depicted in Fig. 6.10. All sensor nodes must be in this uniquely identifiable location; traces are only cut if each trace $PT_i, i = \{1, ..., n\}$ of n sensor nodes indicates that this is the case. One may note that such points in time usually exist, as sensor nodes reside most of their deployment time ($\geq$ 90%) in a low-power mode. In this work, segmentation points are restricted to the points in time where one node enters the uniquely identifiable location. Note that the other node must already be in the uniquely identifiable location as described above.

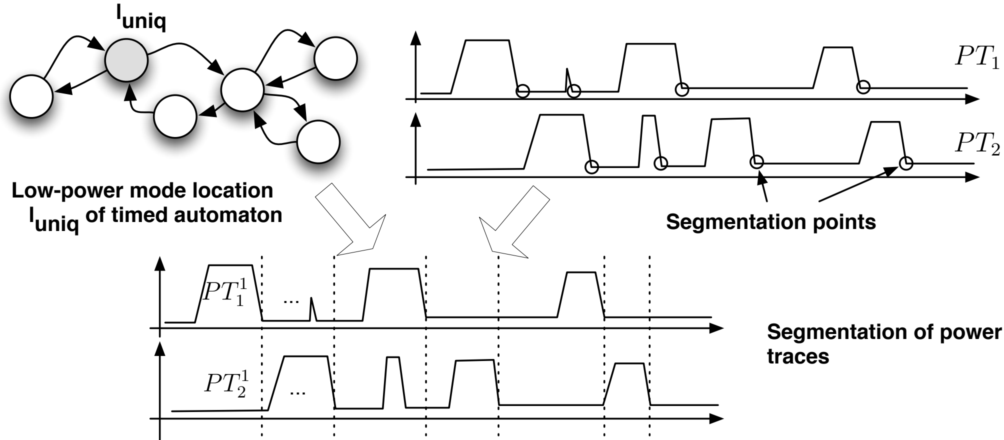

Figure 6.10: Given is a system model with an unambiguously identifiable location l_{uniq} (here in grey) common to each possible execution. In this case, the segmentation of the traces is performed such that each segment starts and ends in this location. The corresponding segmentation of PT_1 and PT_2 is indicated on the upper right: Segmentation points are encircled. The segmentation of PT_1, PT_2 results in segmented trace models TM_1, TM_2.

6.2.4 Segmented power trace testing

We propose a depth-first search approach for testing power trace segments. A single context is generated (if the segment conforms). This context is directly applied to the consecutive segment. This approach is outlined in procedure exploreSegment in Algorithm 6.1 for two power traces models TM_1 and TM_2. It takes the system models and the set of segments $TM^j_{(1,2)}, j \in \{1, ..., m\}$ as inputs, where $m \in \mathbb{N}$ denotes the number of segments. Segments are checked using the Uppaal model checker `verifyta`. `verifyta` is called with a property ϕ to be verified for a model $\mathcal{M}$, i.e., it checks whether $\mathcal{M} \models \phi$ holds. It returns a counterexample, i.e., a context, if the property fails. Hence, Algorithm 6.1 uses the inverted property $A[]\ \texttt{not}(final_1 \wedge final_2)$ to generate a context *newCon*. If the property does hold, `verifyta` does not return a counterexample, i.e., a context does not exist. A context is applied to the models of the next segment using the initialization model *applyContext* (cf. Fig. 6.8).

Segments are checked one at a time. There may be two results of checking whether $A[]\ \texttt{not}(final_1 \wedge final_2)$ holds for a given segment:

- A context *newCon* exists, so the exploration continues with the next segment. If the current segment was the last segment ($cur == m$), the conformance test has succeeded, i.e., the algorithm returns `true`.
- There exists no context. This means that the current segment does not conform given its own context of the previous segment $con[cur-1]$. Hence, the algorithm backtracks to the previous segment. It tries to generate a new context by excluding previously unsuccessful contexts in the query to the model checker. Therefore, it maintains the set of contexts it has already

unsuccessfully explored ($exclude[m]$) for each segment. The algorithm backtracks at most to the first segment. If the model checker cannot find a valid context (anymore) for the first segment, the conformance test fails, i.e., the algorithm returns `false`.

Note that the algorithm works for both a zone-based context and a single discrete context. Algorithm 6.1 is used with discrete contexts in the case study.

Algorithm 6.1 Procedure EXPLORESEGMENT iteratively checks power trace model segments for two sensor nodes. *Inputs* are the number of segments m, the trace model segments $\{TM_1^1, \ldots, TM_1^m\}, \{TM_2^1, \ldots, TM_2^m\}$, and the two system models Sys_1, Sys_2. $applyContext(con)$ is an Uppaal model to apply a context con as the initial state for the system models. *initialContext* is the initial state of the system models before the conformance test. The *output* is a Boolean for indicating conformance of the power trace models to the specification models.

```
procedure EXPLORESEGMENT(m, {TM_1^1, ..., TM_1^m}, {TM_2^1, ..., TM_2^m}, Sys_1, Sys_2,
    applyContext, initialContext){
exclude[m] = {∅, ..., ∅}
con[m] = {∅, ..., ∅}
con[0] = initialContext
cur = 1
while cur > 0 do
    newCon = verifyta(Sys_1||TM_1^cur||Sys_2||TM_2^cur||applyContext(con[cur − 1]) ⊨
        A[] not(final_1 ∧ final_2 ∧ not exclude[cur]))
    if newCon exists then
        if cur == m then
            return true /* the conformance test succeeded */
        else
            con[cur] = newCon /* context applied to nextSegment */
            cur = cur + 1 /* proceed to next Segment */
        end if
    else
        cur = cur − 1 /* backtrack to previous segment */
        exclude[cur] = exclude[cur] ∪ con[cur]
    end if
end while
return false /* the conformance test failed */
}
```

6.3 Case Study

In this case study, the applicability of the proposed segmentation of power trace models is investigated. As the main goal, it is tested whether

$Sys_1||TM_1||Sys_2||TM_2 \models E <> (final_1 \wedge final_2)$ holds. One may already note that the conformance of two interacting nodes with their simultaneously measured power consumptions cannot be verified by Uppaal in a single execution (cf. Sec. 6.2.1). This is because Uppaal allocates a large amount of memory that exceeds the capabilities of typically employed commodity computer.

6.3.1 Testing communicating sensor nodes

A fundamental property of sensor networks is their low-power operation. Since the radio is the major contributor to power consumption, the focus of this case study is on the MAC layer. MAC protocols trade off bandwidth for energy by duty-cycling the radio. The MAC protocol used in the case study is selected from the predominant class of random-access MAC protocols. When testing such MAC protocols, one focuses on a small set of neighboring nodes. In particular, most testcases can be formulated with two or three sensor nodes. A sender and a receiver are fundamental. A third node may be added to test for interference, hidden terminals, etc.. This case study tests for basic functionality of the MAC protocol using two sensor nodes running the Harvester application (cf. Sec. 2.2). The models from Chapter 5 are refined for better fidelity in testing the interaction. Although Harvester was previously discussed, some important concepts are recapitulated.

Harvester features a LPL MAC. This implies that a node may only receive at certain times, when its radio is turned on. The time between two consecutive wake-ups is called the wake-up interval T_W. In particular, Harvester uses a variant of a synchronized low-power MAC protocol as described in Sec. 2.2.1. In this MAC protocol scheme, nodes sleep for most of the time, yet wake up and turn on their radio after a given wake-up period to check for ongoing traffic. The MAC protocol offers two distinct operations to the sensor node software: (a) broadcast transmissions that are used to send a message to all nodes in the neighborhood and (b) unicast transmissions, where a node sends a message to a specific neighbor. In steady-state operation, nodes only send a unicast message to another node shortly before this other node wakes up to be ready for reception. A node stops its unicasts immediately after receiving an acknowledgement from the addressed node. Broadcasts address all neighbors and are sent therefore for the complete wake-up period T_W in order to guarantee that all nodes in the neighborhood receive this message. Hence, a broadcast takes considerably longer than a unicast, since it needs to last a complete wake-up period.

This allows the following high-level modeling:

- A wake-up window of $T_w = [0.96s, 1s]$ is defined. A node must wake up each T_W.
- A sender wanting to send a unicast packet is synchronized to its receiver.

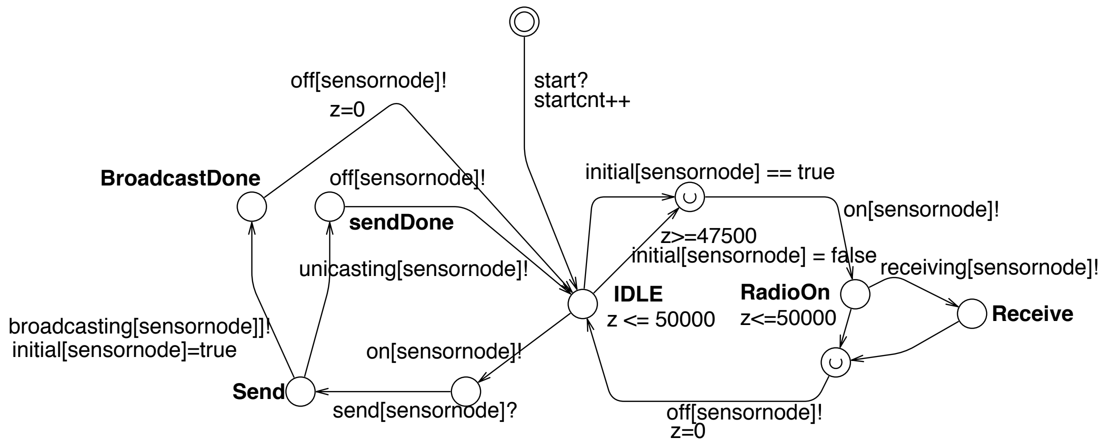

Figure 6.11: Uppaal radio software model. It includes periodically listening (on the right) and support for broadcast and unicast operations (on the left).

- A sender may start a broadcast at any time and sends for a complete T_w.
- After any broadcast the wake-up cycle of the sending node may shift in time.
- After any radio operation, i.e., sending, receiving or listening for traffic, a node goes back to sleep.

The underlying radio and the basic MAC functionality is implemented in a radio component model, including the timing of receive, unicast and broadcast operations as shown in Fig. 6.12. The higher level functionality of periodic wake-ups and re-synchronization after transmissions is modeled in a radio software model. Figure 6.11 displays the refined software model that may send broadcast and unicast packets using channels `broadcasting` and `unicasting`. Both models feature an initialization synchronization `start?` as previously described in Fig. 6.8. Additionally they exploit the domain-specific abstraction of the recurrent state and feature a single initial location corresponding to the low-power mode.

Figure 6.12: Uppaal hardware component model comprising the radio hardware and basic MAC functionality with receive, unicast and broadcast operations.

6.3.2 Experimental setup

In the case study, two communicating sensor nodes are monitored: one that only sends unicast or broadcast messages (node 10) and the other only receiving (node 12). These sensor nodes are fed by a constant voltage from a DC Power Supply (Agilent E3631A). Two channels, one per sensor node, of a Tektronix MSO4054 Mixed Signal Oscilloscope are used for sampling power consumption at a rate of 50kS/s, i.e., every $20\mu s$. The low current draw of the sensor node combined with the reduced fidelity of the oscilloscope for small measurements necessitates an amplification of the sensor nodes current draw. We choose a Maxim MAX9922 Evaluation kit and change the Sense resistor to 1Ω to be comparable to the experiments in Chapter 5. All runs were performed on a Sun-Fire-X2200-M2-64 blade running Linux. It features 2 dual core 64-bit AMD Opteron processors, i.e. 4 cores, running at 2.6 GHz. The blade has 8 GB RAM. The command-line verifier of Uppaal 4.1.2 is used (`verifyta`). In particular, `verifyta` is run with the `-u` option to obtain information about explored and stored states.

6.3.3 Results

There are two testcases that include the wake-up behavior of two nodes. The first testcase (Unicast) includes a single unicast message as depicted in Fig. 6.13. The second testcase (Broadcast) features a single broadcast as depicted in Fig. 6.14. Other than these communication events, both nodes only perform periodic wake-ups for listening. Table 6.1 characterizes the measured power traces and summarizes the experiments for both testcases. We can see that the combined model always fails due to an "Out of memory"-exception (having a high exploration load of 7,520,099 or 6,942,903 states) and hence is not applicable, even for simple testcases.

The right part of Table 6.1 shows that using the segmentation approach the traces are partitioned into 15 or 8 segments. By segmenting the power traces, the number of locations per segment is limited. In turn, the number of states per segment is limited. The maximal number of stored states is 1,881,237 states for the Unicast (for 88 locations in TM_1^8 and 4 locations in TM_2^8) and $2,614,433$ for the Broadcast testcase (for 6265 locations in TM_1^2 and 52 locations in TM_2^2, the broadcast can be seen in Fig. 6.14). This allows PTT for sequentially testing the synchronously measured power traces. The limited number of states for individual segments renders the method more tolerant with respect to state-space explosion. The runtime for testing the conformance of the Unicast testcase is $8597s$; for the Broadcast testcase, the runtime is $5218s$.

Fig. 6.15 depicts the results of the segmentation approach, where each data point represents a segment ('x' for the Unicast and 'o' for the Broadcast testcase) with respect to its number of locations of TM_1 and the number of stored states by the model checker. The number of locations visited within TM_2 is not

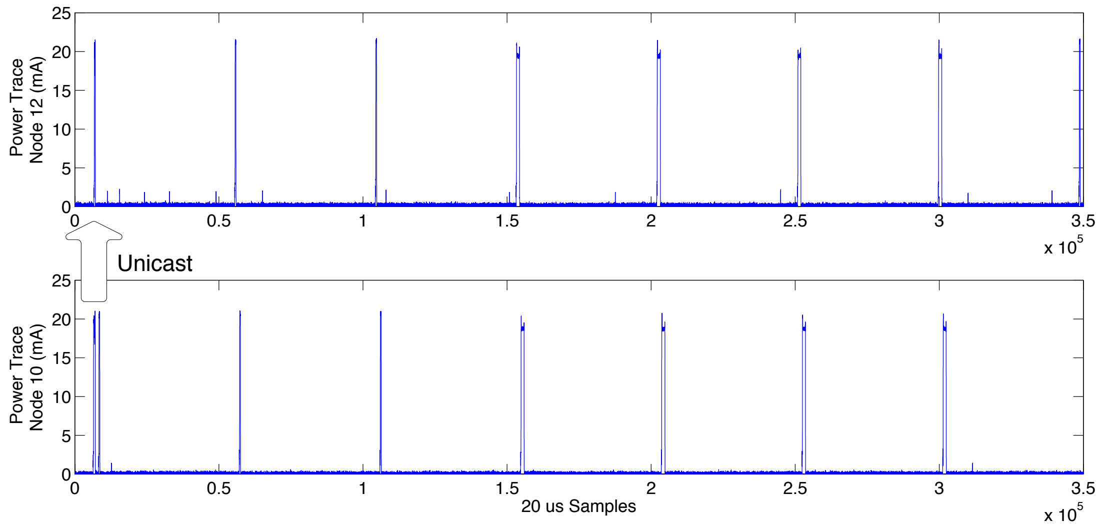

Figure 6.13: Unicast testcase: Power trace for a scenario with a single unicast transmitted from node 10 to node 12.

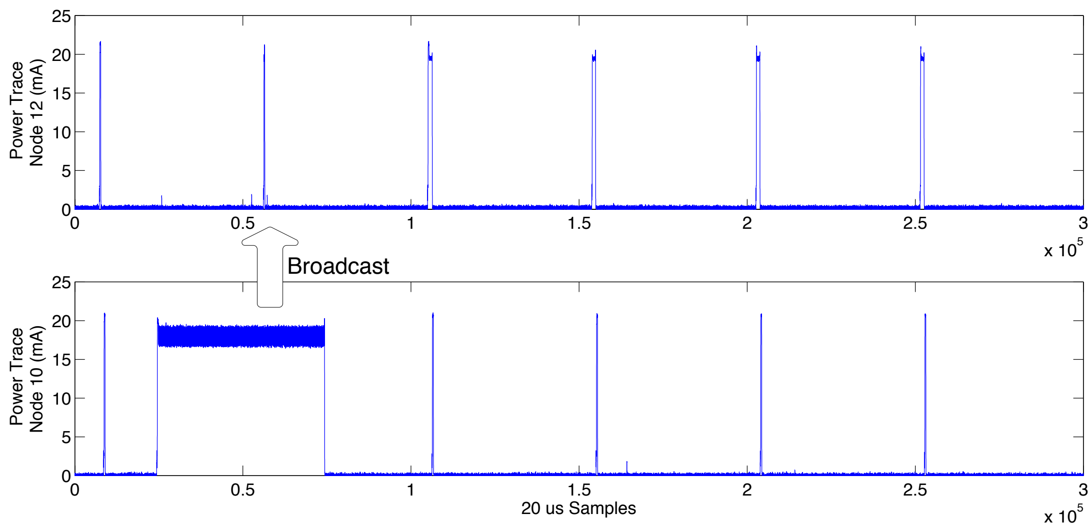

Figure 6.14: Broadcast testcase: Power trace for a scenario with a single broadcast transmitted from node 10 to node 12.

reported. There is a single outlier in the figure on the right: the segment where a broadcast is performed. It features many locations. Due to a low degree of non-determinism in this case, the corresponding trace models do not induce a large number of visited state.

The lines in the figure repeat the results for the naive verification from Fig. 6.4 for comparison. Using the naive verification approach, the exploration aborts after less than 40 locations visited within the power trace model TM_1. Fig. 6.15 also depicts the memory wall, an experimental barrier for the naive verification of $Sys_1 || TM_1 || Sys_2 || TM_2 \models (E <> final_1 \wedge E <> final_2)$. It is this

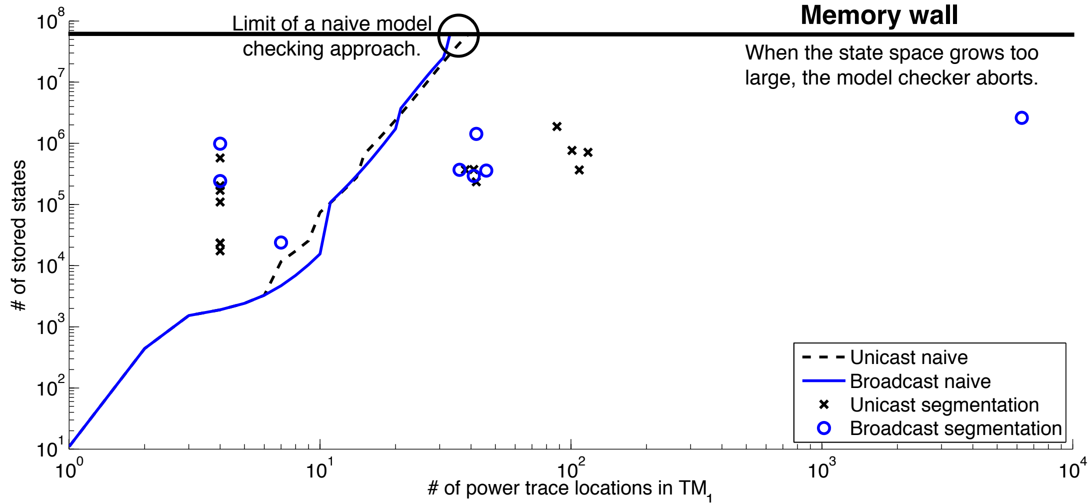

Figure 6.15: Log-log plot of the number of states stored by Uppaal versus the number of locations in TM_1 for both testcases.

memory wall that limits the number of state visits in a single run of the model checker and that prohibits a scaling of PTT to more complex models. However, due to the segmentation of the power traces as suggested in this chapter, the individual executions of the model checker stay well below this memory wall. It is this segmentation approach that allows PTT to be used for more complex models.

Model	Naive		Segmented			
	Locations	States	Locations	States	Segments	Σ States
Unicast	1213	Fails	≤217	≤1,881,237	15	5,828,634
Broadcast	6957	Fails	≤6317	≤2,614,433	8	6,318,540

Table 6.1: Comparing power trace locations (Locations) and the states stored (States) for two testcases.

6.4 Related work

The idea of a segmentation of the state space is related to previous work on stateless model checking [God97]. Stateless model checkers use a stateless depth-first search of the state space, i.e., visited states are not permanently stored. They verify safety properties, such as reachability of a given state. Generally, the stateless approach may result in non-termination of the exploration in case of cycles. It may also lead to multiple redundant explorations of unstored parts of the state space. These drawbacks can be avoided for

power traces with linear behavior and the presented approach of segmentation. More closely related is the work of Dwyer et al. [DDH03] on quasi-cyclic systems. Quasi-cyclic systems always return to a given state (set) that features a fixed valuations for (a subset of) variables. This is equivalent to the discussed domain-specific property of low-power embedded systems that always return to the low-power mode. Different to our presented approach based on the linear power trace properties, Dwyer et al.'s approach may also lead to redundant explorations of the state space. Additionally, both approaches are targeted for untimed models, while PTT is focussed on timed automata.

6.5 Summary

To make PTT feasible for concurrent systems, such as a set of communicating sensor nodes, a naive approach using a model checker does not work. The issue is that the model checker cannot exploit the inherent linear behavior of power trace models. In turn, it was shown that power traces can be segmented. This allows PTT to partition the reachability problem into smaller, manageable segments. This segmentation relies on the fact that for linear traces, states do not have to be maintained (indefinitely). In order to check the conformance, the context from each segment has to be considered when checking the consecutive segment. The context can be determined by utilizing counterexamples provided by the model checker. The segmentation proposed in this chapter exploits that there is a uniquely identifiable low-power location vector for low-power systems - the low-power mode. The chapter presented a sound and complete algorithm for sequentially checking power trace segments. While the presented procedure is merely a first step towards an efficient PTT approach, it clearly shows that domain-specific adaptations are a powerful mechanism in order to allow the industrial application of state-based methods beyond pure functional verification.

7

Conclusions

This chapter summarizes the contributions of this thesis and discusses potential directions for future research.

7.1 Contributions

System testing of WSNs is a complex task. This thesis contributes several solutions for testing WSNs. Its goal is to provide automated tool support for executing testcases, extracting meaningful information from test executions, analyzing the monitored information, and checking for the conformance to a specification of expected behavior. In the following, the main contributions for testing sensor networks are presented topically grouped.

- **Modeling**
  This thesis described different modeling techniques that can be exploited for the analysis of a WSN. In Chapter 2, a stochastic model of communication was presented that can be used to analyze deployments with respect to the placement of sensor nodes. It allows engineers to optimize their node deployments using Multiobjective Evolutionary Algorithms to achieve a well-connected, robust network of nodes. Chapter 5 presented timed automata models for sensor nodes. By annotating timing behavior and power consumption, the models can be used to automatically analyze measured power consumption using a novel conformance test. These timed automata models have potential for further use as described in the future work below.

- **Test platforms**
  A second part of the thesis concerned test platforms for sensor networks. Two specific open questions were addressed: Can there can be an integration of several test platforms into a comprehensive framework? How can the test of sensor networks be automated for testing and monitoring functional and non-functional properties of a sensor network over the development period? Chapter 3 described an approach for multi-platform testing based on test platform adaptors. Chapter 4 presented a test architecture allowing engineers to continuously test their application and monitor functional and non-functional properties.

- **Analysis of system executions**
  An important part of testing is the analysis of test executions. In this thesis, two different methods were presented: the first method concerned the analysis of functional properties; the second method concerned non-functional properties, or more specifically power consumption. The Rupeas language uses an event abstraction for analyzing log files. As Chapter 3 showed, Rupeas can be used to analyze log files from test executions, e.g., to analyze for detecting defects in the MAC or routing protocol. Chapter 5 and 6 presented Power Trace Testing (PTT). PTT automatically tests the power consumption of a system. While the particular focus in this thesis was PTT's application to WSNs, it is generally applicable for the analysis of any system. In fact, it is applicable for many physical properties, irrespective if they are inputs to or outputs of a system. The only requirement is that the physical quantity only depends on or affects the current state of the system. Chapter 5 presented formal foundations of PTT's conformance test and Chapter 5 and 6 described various optimizations to make it applicable for a real sensor network application.

The resulting tool implementations have been applied to test sensor network application. As shown in several case studies, the tools find errors in implementations and support the debugging process.

7.2 Future work

The work on testing WSNs, as presented in this thesis, has stimulated research in several directions that bears potential for future work.

- **Flocklab and PTT**
  To accommodate for distributed power measurements and control of

the tested sensor node, a new testbed was developed called Flocklab [BLM+09]. A particular feature of Flocklab is the capability to continuously log power measurements and internal state via GPIO pins. Flocklab was designed to apply the presented PTT in a distributed context, i.e., on interacting sensor nodes. For supporting the adoption of PTT for the typical user, an easier description format for the specification of sensor nodes, their components' power consumption and the protocol stack, would be beneficial. Additionally, the system model used in PTT can be easily extended to concurrently test measurements of power consumption and (binary) observations of internal state via GPIO pins.

- **Test generation**
  In this work, various methods were presented that can be applied in testing for the analysis of executions such as event analysis in Chapter 3 and PTT in Chapter 5. A subject for future work is the automatic generation of testcases. In particular, the formal models as introduced for PTT can be exploited for testcase generation similar to TRON [LMN04, HLM+08, KGL09]. However, testcase generation relies on an association of high-level modeling with the implementation on a sensor node. Further research needs to address methods to either generate code from high-level models, e.g., on a virtual machine, or automatically generate test adaptor code for interacting with the sensor nodes. As such the integration of design and testing towards "Design For Validation" is much needed.

- **Runtime monitoring**
  A prime concern of this thesis is to ensure that a WSN operates correctly when it is deployed. Nevertheless, experience has shown that deployed systems still contain a non-negligible number of defects. Hence, runtime systems are needed that detect problems of a running system and may even try to correct them. For the online monitoring problem, some initial work was performed in [dJWL09] using a model-based diagnosis approach. Other work investigated at the use of assertions [RM09], overhearing [MKL+07, RRV07] and application-specific solutions [RCK+05, RB06, MWW+09], yet a general runtime monitoring solution is missing. Moreover, if problems are detected, software may determine some action to alleviate the problem. A first approach using software rejuvenation was performed in [WML10], inspired by [CGK+09]. It shows that rebooting individual software components instead of a complete node improves system availability. This initial work highlights that further research should investigate the application of software rejuvenation and software healing approaches for sensor networks.

Bibliography

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AM04] Rajeev Alur and P. Madhusudan. Decision problems for timed automata: A survey. In *In Proceedings of SFM'04, Lect. Notes Comput. Sci. 3185, 1–24*, pages 1–24. Springer, 2004.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[BDL+07] Jan Beutel, Matthias Dyer, Roman Lim, Christian Plessl, Matthias Woehrle, Mustafa Yuecel, and Lothar Thiele. Demo abstract: Automated wireless sensor network testing. In *Proc. 4th Int'l Conf. on Networked Sensing Systems (INSS 2007)*, page 303. IEEE, June 2007.

[Bei90] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., 1990.

[BGH+09] Jan Beutel, Stephan Gruber, Andreas Hasler, Roman Lim, Andreas Meier, Christian Plessl, Igor Talzi, Lothar Thiele, Christian Tschudin, Matthias Woehrle, and Mustafa Yuecel. PermaDAQ: A scientific instrument for precision sensing and data recovery in environmental extremes. In *Proc. 8th ACM/IEEE Int'l Conf. on Information Processing in Sensor Networks (IPSN 2009)*, pages 265–276. ACM/IEEE, April 2009.

[BISV08] Guillermo Barrenetxea, François Ingelrest, Gunnar Schaefer, and Martin Vetterli. The hitchhiker's guide to successful wireless sensor network deployments. In *Proc. 6th ACM Conf. Embedded Networked Sensor Systems (SenSys 2008)*, 2008.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[BKX+06] Xiaole Bai, Santosh Kuma, Dong Xua, Ziqiu Yun, and Ten H. La. Deploying wireless sensors to achieve both coverage and connectivity. In *MobiHoc '06: Proceedings of the 7th ACM international symposium on Mobile ad hoc networking and computing*, pages 131–142. ACM Press, 2006.

[BLM+09] Jan Beutel, Roman Lim, Andreas Meier, Lothar Thiele, Christoph Walser, Matthias Woehrle, and Mustafa Yuecel. Poster abstract: The FlockLab Testbed Architecture. In *Proc. 7th ACM Conf. Embedded Networked Sensor Systems (SenSys 2009)*, pages 415–416, November 2009.

[BLR05] Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Priced timed automata: Algorithms and applications. In *Proc. of Formal Methods for Components and Objects (FMCO'04)*, pages 162–182, 2005.

[BLTZ03] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler. PISA—A Platform and Programming Language Independent Interface for Search Algorithms. In *Conference on Evolutionary Multi-Criterion Optimization (EMO 2003)*, volume 2632 of *LNCS*, pages 494–508, 2003.

[Bou07] Athanassios Boulis. Castalia: revealing pitfalls in designing distributed algorithms in wsn. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 407–408. ACM, 2007.

[BRWR10] Jan Beutel, Kay Roemer, Matthias Woehrle, and Matthias Ringwald. Deployment Techniques for Sensor Networks. In *Sensor Networks - Where Theory Meets Practice*, pages 219–248. Springer, 2010.

[BS08a] Henrik Bohnenkamp and Mariëlle Stoelinga. Quantitative testing. In *EMSOFT '08: Proceedings of the 8th ACM international conference on Embedded software*, pages 227–236. ACM, 2008.

[BS08b] Timothy Bourke and Arcot Sowmya. Automatically transforming and relating uppaal models of embedded systems. In *EMSOFT '08: Proceedings of the 8th ACM international conference on Embedded software*, pages 59–68. ACM, 2008.

[BY04] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In W. Reisig and G. Rozenberg, editors, *In Lecture Notes on Concurrency and Petri Nets*, Lecture Notes in Computer Science vol 3098. Springer–Verlag, 2004.

[BYAH06] Michael Buettner, Gary V. Yee, Eric Anderson, and Richard Han. X-mac: a short preamble mac protocol for duty-cycled wireless sensor networks. In *Proc. 4th ACM Conf. Embedded Networked Sensor Systems (SenSys 2006)*, pages 307–320. ACM Press, 2006.

[CBB+06] Yu-Chung Cheng, John Bellardo, Péter Benkö, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Jigsaw: solving the puzzle of enterprise 802.11 analysis. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 39–50. ACM, 2006.

[CGK+09] Yang Chen, Omprakash Gnawali, Maria Kazandjieva, Philip Levis, and John Regehr. Surviving sensor network software faults. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 235–246. ACM, 2009.

[CLC04] David Cohen, Mikael Lindvall, and Patricia Costa. An introduction to agile methods. *Advances in Computers*, 62:2–67, 2004.

[CLWL06] J.I. Choi, J.W. Lee, M. Wachs, and P. Levis. Opening the sensornet black box. Technical Report SING-06-03, Stanford Information Networks Group, Stanford University, CA, 2006.

[Com08a] Computer Engineering and Networks Lab - ETH Zürich. Cruisecontrol at tik42x.ee.ethz.ch, April 2008.

[Com08b] Computer Engineering and Networks Lab - ETH Zürich. Harvester in tinyos 2 contrib, 2008.

[Cru10] CruiseControl. Cruisecontrol home, April 2010.

[CVS+07] Peter Corke, Philip Valencia, Pavan Sikka, Tim Wark, and Les Overs. Long-duration solar-powered wireless sensor networks. In *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors*, pages 33–37. ACM, 2007.

[DBT+07] M. Dyer, J. Beutel, L. Thiele, T. Kalt, P. Oehen, K. Martin, and P. Blum. Deployment support network - a toolkit for the development of WSNs. In *Proc. 4th European Workshop on Sensor Networks (EWSN 2007)*, pages 195–211, 2007.

[DCG08] F. Zhao; J. Liu D. Chu and M. Goraczko. Que: A sensor network rapid prototyping tool with application experiences from a data center deployment. In *Proc. 5th European Workshop on Sensor Networks (EWSN 2008)*, 2008.

[DCI02] S. Dhillon, K. Chakrabarty, and S. Iyengar. Sensor placement for grid coverage under imprecise detections. In *Proc. 5th Intl. Conf. on Information Fusion*, volume 2, pages 1581–1587, 2002.

[DDH03] Matthew B. Dwyer, William Deng, and John Hatcliff. Space reductions for model checking quasi-cyclic systems. In *Proc. of the 3rd Int'l Conference on Embedded Software*, pages 173–189. Springer, 2003.

[Deb01] Kalyanmoy Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley, 2001.

[DGV04] A. Dunkels, B. Grönvall, and T Voigt. Contiki – a lightweight and flexible operating system for tiny networked sensors. In *Proc. 1nd IEEE Workshop on Embedded Networked Sensors (EmNetS-I)*, pages 455–462, 2004.

[dJWL09] Adriaan de Jong, Matthias Woehrle, and Koen Langendoen. MoMi - model-based diagnosis middleware for sensor networks. In *Proc. 4th Int'l Conf. Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks (MidSens 2009)*, pages 19–24. Springer, December 2009.

[DOTH07] Adam Dunkels, Fredrik Osterlind, Nicolas Tsiftes, and Zhitao He. Software-based on-line energy estimation for sensor nodes. In *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors*, pages 28–32, 2007.

[EAR+06] Emre Ertin, Anish Arora, Rajiv Ramnath, Vinayak Naik, Sandip Bapat, Vinod Kulathumani, Mukundan Sridharan, Hongwei Zhang, Hui Cao, and Mikhail Nesterenko. Kansei: a testbed for sensing at scale. In *IPSN '06: Proceedings of the 5th international conference on Information processing in sensor networks*, pages 399–406. ACM, 2006.

[EHD04] A. El-Hoiydi and J.D. Decotignie. WiseMAC: An ultra low power MAC protocol for multi-hop wireless sensor networks. In *Proc. 1st Int'l Workshop Algorithmic Aspects of Wireless Sensor Networks (ALGOSENSORS 2004)*, pages 18–31, 2004.

[FDLS08] Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. Quanto: Tracking energy in networked embedded systems. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, pages 323–338, December 2008.

[Fre08] Goran Frehse. Phaver: algorithmic verification of hybrid systems past hytech. *Int. J. Softw. Tools Technol. Transf.*, 10(3):263–279, 2008.

[GFJ+09] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection tree protocol. In *SenSys '09: Proceedings of the 7th ACM conference on Embedded network sensor systems*, pages 1–14. ACM, 2009.

[God97] Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186. ACM, 1997.

[Hen96] Thomas A. Henzinger. The theory of hybrid automata. In *Proc. 11th Annual Symposium on Logic in Computer Science (LICS)*, pages 278–292. IEEE Computer Society Press, 1996.

[HHP+08] I.J. Haratcherev, G.P. Halkes, T.E.V. Parker, O.W. Visser, and K.G. Langendoen. PowerBench: A scalable testbed infrastructure for benchmarking power consumption. In *Int. Workshop on Sensor Network Engineering (IWSNE)*, pages 37–44, June 2008.

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. Hytech: A model checker for hybrid systems. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 460–463. Springer-Verlag, 1997.

[HKWW06] Vlado Handziski, Andreas Koepke, Andreas Willig, and Adam Wolisz. Twist: a scalable and reconfigurable testbed for wireless indoor experiments with sensor networks. In *Proc. 2nd international workshop on Multi-hop ad hoc networks: from theory to reality (REALMAN '06)*, pages 63–70. ACM Press, 2006.

[HLM+08] Anders Hessel, Kim G. Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using UPPAAL. In *Formal Methods and Testing*, volume 4949/2008 of *Lecture Notes in Computer Science*, pages 77–117. Springer, 2008.

[HMA+08] Naofumi Homma, Atsushi Miyamoto, Takafumi Aoki, Akashi Satoh, and Adi Shamir. Collision-based power analysis of modular exponentiation using chosen-message pairs. In *10th Int'l Workshop*

on Cryptographic Hardware and Embedded Systems (CHES), pages 15–29. Springer, August 2008.

[HMP92] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. What good are digital clocks? In *ICALP '92: Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, pages 545–558. Springer-Verlag, 1992.

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *SIGPLAN Not.*, 35(11):93–104, 2000.

[JDCS07] Xiaofan Jiang, Prabal Dutta, David Culler, and Ion Stoica. Micro power meter for energy monitoring of wireless sensor networks at scale. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 186–195. ACM, 2007.

[JLS00] Henrik Ejersbo Jensen, Kim Guldstrand Larsen, and Arne Skou. Scaling up uppaal automatic verification of real-time systems using compositionality and abstraction. In *FTRTFT '00: Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 19–30. Springer-Verlag, 2000.

[Jou06] D. B. Jourdan. *Wireless Sensor Network Planning with Application to UWB Localization in GPS-Denied Environments*. PhD thesis, Massachusetts Institute of Technology, 2006.

[KAJV07] Charles Edwin Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code (awarded best paper). In *4th Symposium on Networked Systems Design and Implementation (NSDI 2007)*. USENIX, April 2007.

[KGGK06] Andreas Krause, Carlos Guestrin, Anupam Gupta, and Jon Kleinberg. Near-optimal sensor placements: maximizing information while minimizing communication cost. In *Proc. 5th Int'l Conf. Information Processing Sensor Networks (IPSN '06)*, pages 2–10. ACM Press, 2006.

[KGL09] Brian Nielsen Kim G. Larsen, Marius Mikucionis. *Uppaal Tron User Manual*. CISS, BRICS, Aalborg University, Aalborg, Denmark, June 2009.

[KLP+09] Kevin Klues, Chieh-Jan Mike Liang, Jeongyeup Paek, Răzvan Musăloiu-E, Philip Levis, Andreas Terzis, and Ramesh Govindan.

Tosthreads: thread-safe and non-invasive preemption in tinyos. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 127–140. ACM, 2009.

[KNG+04] D. Kotz, C. Newport, R.S. Gray, J. Liu, Y. Yuan, and C. Elliott. Experimental evaluation of wireless simulation assumptions. In *Int'l Workshop Modeling Analysis and Simulation of Wireless and Mobile Systems (MSWiM 04)*, pages 78–82. ACM Press, New York, October 2004.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 2008.

[Lan08] K.G. Langendoen. Medium access control in wireless sensor networks. In H. Wu and Y. Pan, editors, *Medium Access Control in Wireless Networks*, pages 535–560. Nova Science Publishers, Inc., may 2008.

[LBV06] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *Proc. 20th Int'l Parallel and Distributed Processing Symposium (IPDPS 2006)*, pages 8–15, 2006.

[LGH+05] P. Levis, D. Gay, V. Handziski, J.-H.Hauer, B.Greenstein, M.Turon, J.Hui, K.Klues, C.Sharp, R.Szewczyk, J.Polastre, P.Buonadonna, L.Nachman, G.Tolle, D.Culler, and A.Wolisz. T2: A second generation os for embedded sensor networks. Technical Report TKN-05-007, Telecommunication Networks Group, Technische Universität Berlin, November 2005.

[LHL08] M. Lodder, G. Halkes, and K. Langendoen. A global-state perspective on sensor network debugging. In *Proc. 5th IEEE Workshop on Embedded Networked Sensors (HotEmNets 2008)*, pages 37–41, June 2008.

[LLWC03] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proc. 1st ACM Conf. Embedded Networked Sensor Systems (SenSys 2003)*, pages 126–137, November 2003.

[LMN04] Kim Guldstrand Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using uppaal. In *4th Int'l Workshop Formal Approaches to Software Testing (FATES 2004), Revised Selected Papers*, pages 79–94, September 2004.

[LrCC+09] Konrad Lorincz, Bor rong Chen, Geoffrey Werner Challen, Atanu Roy Chowdhury, Shyamal Patel, Paolo Bonato, and Matt Welsh. Mercury: a wearable sensor network platform for high-fidelity motion analysis. In *Proc. 7th Int'l Conf. on Embedded Networked Sensor Systems (SenSys '09)*, pages 183–196, November 2009.

[LWG05] O. Landsiedel, K. Wehrle, and S. Gotz. Accurate prediction of power consumption in sensor networks. In *Proc. 2nd IEEE Workshop on Embedded Networked Sensors (EmNetS-II)*, pages 37–44. IEEE Computer Society, 2005.

[LWMB09] Roman Lim, Matthias Woehrle, Andreas Meier, and Jan Beutel. Poster abstract: Harvester - energy savings through synchronized low-power listening. In *Adjunct Proc. 6th European Workshop on Sensor Networks (EWSN 2009)*, pages 29–30. Springer, February 2009.

[M+07] David Moss et al. Bug in cc2420 timestamp. https://www.millennium.berkeley.edu/pipermail/tinyos-help/2007-October/028901.html, October 2007.

[Mei09] Andreas Meier. *Safety-Critical Wireless Sensor Networks*. PhD thesis, ETH Zurich, June 2009.

[MKAG08] Mohammad Maifi, Hasan Khan, Tarek Abdelzaher, and Kamal Kant Gupta. Towards diagnostic simulation in sensor networks. In *Distributed Computing in Sensor Systems*, pages 252–265. Springer, 2008.

[MKL+07] Mohammad Maifi, Hasan Khan, Liqian Luo, Chengdu Huang, and Tarek Abdelzaher. SNTS: Sensor network troubleshooting suite. In *Distributed Computing in Sensor Systems*, volume Volume 4549/2007, pages 142–157. Springer, 2007.

[MLM+05] Pedro José Marrón, Andreas Lachenmann, Daniel Minder, Jörg Hähner, Robert Sauter, and Kurt Rothermel. TinyCubus: A flexible and adaptive framework for sensor networks. In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN 2005)*, pages 278–289, January 2005.

[MLN04] Marius Mikucionis, Kim G. Larsen, and Brian Nielsen. T-uppaal: Online model-based testing of real-time systems. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 396–397. IEEE Computer Society, 2004.

[MPS+02] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless sensor networks for habitat monitoring. In *1st ACM Int. Workshop on Wireless Sensor Networks and Application (WSNA 2002)*, pages 88–97, 2002.

[MRBT08] Andreas Meier, Tobias Rein, Jan Beutel, and Lothar Thiele. Coping with unreliable channels: Efficient link estimation for low-power wireless sensor networks. In *Proc. 5th Intl Conf. Networked Sensing Systems (INSS 2008)*, pages 19–26. IEEE, June 2008.

[MRWZ06] Ratul Mahajan, Maya Rodrig, David Wetherall, and John Zahorjan. Analyzing the MAC-level behavior of wireless networks in the wild. *SIGCOMM Comput. Commun. Rev.*, 36(4):75–86, 2006.

[MWW+09] Andreas Meier, Matthias Woehrle, Mischa Weise, Jan Beutel, and Lothar Thiele. Nose: Efficient maintenance and initialization of wireless sensor networks. In *Proc. Sixth Annual IEEE Communications Society Conference on Sensor, Mesh, and Ad Hoc Communications and Networks (SECON 2009)*, pages 1–9. IEEE, June 2009.

[Mye79] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., 1979.

[NS07] Nguyet T. M. Nguyen and Mary Lou Soffa. Program representations for testing wireless sensor network applications. In *DOSTA '07: Workshop on Domain specific approaches to software test automation*, pages 20–26. ACM, 2007.

[ODE+06] Fredrik Osterlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-level sensor network simulation with COOJA. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pages 641–648, November 2006.

[OSFC07] D. O'Rourke, Conor Brennan Szymon Fedor, and Martin Collier. Reception region characterisation using a 2.4ghz direct sequence spread spectrum radio. In *Proc. 4th IEEE Workshop on Embedded Networked Sensors (EmNetS-IV)*, 2007.

[Par97] B. Parhami. Defect, fault, error,..., or failure? *IEEE Transactions on Reliability*, 46(4):450–451, December 1997.

[PD07] Andrew Glover Paul Duvall, Steve Matyas. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

[PG07] Jeongyeup Paek and Ramesh Govindan. Rcrt: rate-controlled reliable transport for wireless sensor networks. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 305–319. ACM, 2007.

[PHC04] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Proc. 2nd ACM Conf. Embedded Networked Sensor Systems (SenSys 2004)*, pages 95–107. ACM Press, New York, 2004.

[PY08] Mauro Pezze and Michal Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2008.

[RB06] Stanislav Rost and Hari Balakrishnan. Memento: A health monitoring system for wireless sensor networks. In *Proc. 3rd IEEE Communications Society Conf. Sensor, Mesh and Ad Hoc Communications and Networks (IEEE SECON 2006)*, 2006.

[RCK+05] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *Proc. 3rd ACM Conf. Embedded Networked Sensor Systems (SenSys 2005)*, pages 255–267. ACM Press, New York, 2005.

[Reg07] John Regehr. Safe and structured use of interrupts in real-time and embedded software. In *Handbook of Real-Time and Embedded Systems*. CRC Press, 2007.

[RKW+06] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: detecting the unexpected in distributed systems. In *NSDI'06: Proceedings of the 3rd conference on 3rd Symposium on Networked Systems Design & Implementation*, pages 115–128. USENIX Association, 2006.

[RM04] K. Römer and F. Mattern. The design space of wireless sensor networks. *IEEE Wireless Communications*, 11(6):54–61, December 2004.

[RM09] Kay Römer and Junyan Ma. Pda: Passive distributed assertions for sensor networks. In *IPSN '09: Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, pages 337–348. IEEE Computer Society, 2009.

[RR07] Matthias Ringwald and Kay Römer. Deployment of sensor networks: Problems and passive inspection. In *Proceedings of the 5th Workshop on Intelligent Solutions in Embedded Systems (WISES '07)*, pages 180–193, June 2007.

[RR08] Kay Römer and Matthias Ringwald. Increasing the visibility of sensor networks with passive distributed assertions. In *Proc. 4th Workshop on Real-World Wireless Sensor Networks (REALWSN '08)*, pages 36–40, 2008.

[RRV07] Matthias Ringwald, Kay Römer, and Andrea Vitaletti. Passive inspection of sensor networks. In *Proc. 3rd IEEE Int'l Conf. Distributed Computing in Sensor Systems (DCOSS 2007)*, June 2007.

[RVMM05] R. Rajagopalan, P. K. Varshney, C. K. Mohan, and K. G. Mehrotra. Sensor Placement for Energy Efficient Target Detection in Wireless Sensor Networks: A Multi-objective Optimization Approach. In *Conference on Information Sciences and Systems*, 2005.

[SHC+04] V. Shnayder, M. Hempstead, B. Chen, G. Werner-Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proc. 2nd ACM Conf. Embedded Networked Sensor Systems (SenSys 2004)*, pages 188–200. ACM Press, New York, November 2004.

[SHrC+04] Victor Shnayder, Mark Hempstead, Bor rong Chen, Geoff Werner Allen, and Matt Welsh. Simulating the power consumption of large-scale sensor network applications. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 188–200, 2004.

[ST08] Julien Schmaltz and Jan Tretmans. On conformance testing for timed systems. In *FORMATS '08: Proceedings of the 6th international conference on Formal Modeling and Analysis of Timed Systems*, pages 250–264. Springer-Verlag, 2008.

[Str00] Karsten Strehl. *Symbolic Methods Applied to Formal Verification and Synthesis in Embedded Systems Design*. PhD thesis, ETH Zurich, March 2000.

[SY05] Anthony Man-Cho So and Yinyu Ye. On solving coverage problems in a wireless sensor network using voronoi diagrams. In *Proceedings of the 1st Workshop on Internet and Network Economics (WINE 2005)*, pages 584–593. s-lncs, 2005.

[TCLS08] Arsalan Tavakoli, David Culler, Philip Levis, and Scott Shenker. The case for predicate-oriented debugging of sensornets. In *Proceedings of the 5th Workshop on Hot Topics in Embedded Networked Sensors*, 2008.

[TLP05] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Proc. 4th Int'l Conf. Information Processing Sensor Networks (IPSN '05)*, page 67, 2005.

[Tre94] Jan Tretmans. A formal approach to conformance testing. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI*, pages 257–276. North-Holland Publishing Co., 1994.

[Tre08] Jan Tretmans. Model based testing with labelled transition systems. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

[vO06] Michiel van Osch. Hybrid input-output conformance and test generation. In *Formal Approaches to Software Testing and Runtime Verification*, pages 70–84, 2006.

[WALJ+06] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. Fidelity and yield in a volcano monitoring sensor network. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 381–396. USENIX Association, 2006.

[WASW05] G. Werner-Allen, P. Swieskowski, and M. Welsh. MoteLab: A wireless sensor network testbed. In *Proc. 4th Int'l Conf. Information Processing Sensor Networks (IPSN '05)*, pages 483–488, April 2005.

[WBH07] Matthias Woehrle, Dimo Brockhoff, and Tim Hohm. A new model for deployment coverage and connectivity of wireless sensor networks. Technical Report 278, Computer Engineering and Networks Laboratory, ETH Zurich, September 2007.

[WBHB08] Matthias Woehrle, Dimo Brockhoff, Tim Hohm, and Stefan Bleuler. Investigating coverage and connectivity trade-offs in wireless sensor networks: The benefits of MOEAs. In *19th Int'l Conf. on Multiple Criteria Decision Making (MCDM 2008)*, pages 211–221, 2008.

[WBLT08] Matthias Woehrle, Jan Beutel, Roman Lim, and Lothar Thiele. Power monitoring and testing in wireless sensor network development. In *Workshop on Energy in Wireless Sensor Networks, Adj. Proc. DCOSS 2008*, pages IV–3–IV–9, June 2008.

[WBT08] Matthias Woehrle, Jan Beutel, and Lothar Thiele. The system development lifecycle – learning from a sensornet review. Technical Report 283, Computer Engineering and Networks Laboratory, ETH Zurich, May 2008.

[WBYT08] Matthias Woehrle, Jan Beutel, Mustafa Yuecel, and Lothar Thiele. Approaching wireless sensor networks using systematic testing strategies. Technical Report 284, Computer Engineering and Networks Laboratory, ETH Zurich, May 2008.

[WLT09] Matthias Woehrle, Kai Lampka, and Lothar Thiele. Exploiting timed automata for conformance testing of power measurements. In *7th Int'l Conf. on Formal Modelling and Analysis of Timed Systems (FORMATS 2009)*, pages 275–290. Springer, September 2009.

[WLT10] Matthias Woehrle, Kai Lampka, and Lothar Thiele. Conformance testing for cyber-physical systems (available at http://www.tik.ee.ethz.ch/~woehrlem/tik/pub/woehrle.pdf). page (in submission), April 2010.

[WML10] Matthias Woehrle, Andreas Meier, and Koen Langendoen. On the potential of software rejuvenation for long-running sensor network deployments (position paper). In *Proc. 1st Int'l Workshop on Software Engineering for Sensor Network Applications (SESENA 2010)*, pages 44–48, 2010.

[WPBT07] Matthias Woehrle, Christian Plessl, Jan Beutel, and Lothar Thiele. Increasing the reliability of wireless sensor networks with a distributed testing framework. In *Proc. 4th Workshop on Embedded Networked Sensors (EmNets 2007)*, pages 93–97. ACM Press, New York, June 2007.

[WPL+08] Matthias Woehrle, Christian Plessl, Roman Lim, Jan Beutel, and Lothar Thiele. EvAnT: Analysis and checking of event traces for wireless sensor networks. In *Proc. IEEE Int'l Conf. on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC 2008)*, pages 201–208. IEEE, June 2008.

[WPT09] Matthias Woehrle, Christian Plessl, and Lothar Thiele. Poster abstract: Rupeas - an event analysis language for wireless sensor network traces. In *Adjunct Proc. 6th European Workshop on Sensor Networks (EWSN 2009)*, pages 19–20, February 2009.

[WPT10] Matthias Woehrle, Christian Plessl, and Lothar Thiele. Rupeas: Ruby Powered Event Analysis DSL. In *Proc. 7th Int'l Conf. on*

Networked Sensing Systems (INSS 2010), pages 245–248. IEEE, June 2010.

[WXZ+03] Xiaorui Wang, Guoliang Xing, Yuanfang Zhang, Chenyang Lu, Robert Pless, and Christopher Gill. Integrated coverage and connectivity configuration in wireless sensor networks. In *Proc. 1st ACM Conf. Embedded Networked Sensor Systems (SenSys 2003)*, pages 28–39. ACM Press, 2003.

[XRC+04] N. Xu, S. Rangwala, K.K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. In *Proc. 2nd ACM Conf. Embedded Networked Sensor Systems (SenSys 2004)*, pages 13–24, 2004.

[Yov98] Sergio Yovine. Model checking timed automata. In *Lectures on Embedded Systems, European Educational Forum, School on Embedded Systems*, pages 114–152. Springer-Verlag, 1998.

[ZFWT10] Marco Zimmerling, Federico Ferrari, Matthias Woehrle, and Lothar Thiele. Poster abstract: Exploiting protocol models for generating feasible communication stack configurations. In *Proc. ACM/IEEE Int'l Conf. on Information Processing in Sensor Networks (IPSN 2010)*, pages 380–381, April 2010.

[ZHKS06] Gang Zhou, Tian He, Sudha Krishnamurthy, and John A. Stankovic. Models and solutions for radio irregularity in wireless sensor networks. *ACM Trans. Sen. Netw.*, 2(2):221–262, 2006.

[ZK04] E. Zitzler and S. Künzli. Indicator-Based Selection in Multiobjective Search. In *Conference on Parallel Problem Solving from Nature (PPSN VIII)*, volume 3242 of *LNCS*, pages 832–842. Springer, 2004.

[ZK07] Marco Zuniga Zamalloa and Bhaskar Krishnamachari. An analysis of unreliability and asymmetry in low-power wireless links. *ACM Trans. Sen. Netw.*, 3(2):7, 2007.

[Zun04] B. Zuniga, M.; Krishnamachari. Analyzing the transitional region in low power wireless links. In IEEE, editor, *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on, Vol.*, pages 517–526, 2004.

A

Acronyms

WSN Wireless Sensor Network

SUT System Under Test

MCU Microcontroller Unit

RAM Random Access Memory

kbps kilobit per second

MAC Medium Access Control

CTP Collection Tree Protocol

CSMA Carrier Sense Multiple Access

LPL Low-Power Listening

RF Radio Frequency

ISM Industrial, Scientific and Medical

DSL Domain Specific Language

DSN Deployment Support Network

UART Universal Asynchronous Receiver/Transmitter

PTA Power Test Architecture

ADC Analog-to-Digital Converter

CI Continuous Integration

PTT Power Trace Testing

GCI Greatest Common Interval

MOEA Multiobjective Evolutionary Algorithm

B

List of Publications

The following list summarizes the publications that constitute the basis of this thesis. The corresponding chapters are indicated in parentheses.

Matthias Woehrle and Christian Plessl and Jan Beutel and Lothar Thiele. **Increasing the Reliability of Wireless Sensor Networks with a Distributed Testing Framework**. *Proc. 4th Workshop on Embedded Networked Sensors (EmNets 2007), June 2007* (Chapter 3)

Jan Beutel and Matthias Dyer and Roman Lim and Christian Plessl and Matthias Woehrle and Mustafa Yuecel and Lothar Thiele. **Demo Abstract: Automated Wireless Sensor Network Testing**. *Proc. 4th Int'l Conference on Networked Sensing Systems (INSS 2007), June 2007.* (Chapter 3)

Matthias Woehrle and Dimo Brockhoff and Tim Hohm. **A new model for deployment coverage and connectivity of Wireless Sensor Networks**. *Computer Engineering and Networks Laboratory, ETH Zurich, Tecreport 278, September 2007* (Chapter 2)

Matthias Woehrle and Dimo Brockhoff and Tim Hohm and Stefan Bleuler. **Investigating Coverage and Connectivity Trade-offs in Wireless Sensor Networks: The Benefits of MOEAs**. *19th Int'l Conference on Multiple Criteria Decision Making (MCDM 2008), January, 2008.* (Chapter 2)

Matthias Woehrle and Jan Beutel and Roman Lim and Lothar Thiele. **Power monitoring and testing in Wireless Sensor Network Development**. *Workshop on Energy in Wireless Sensor Networks (WEWSN 2008), Adj. Proc. DCOSS, June 2008.* (Chapter 4)

Matthias Woehrle and Christian Plessl and Roman Lim and Jan Beutel and Lothar Thiele. **EvAnT: Analysis and Checking of event traces for Wireless Sensor Networks**. *Proc. IEEE Int'l Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC 2008), June 2008.* (Chapter 3)

Matthias Woehrle and Jan Beutel and Lothar Thiele. **Wireless Sensor Networks Test and Validation**. *Chapter in Handbook of Networked Embedded Systems, 2009.* (Chapter 4)

Matthias Woehrle and Christian Plessl and Lothar Thiele. **Poster Abstract: Rupeas - An Event Analysis Language for Wireless Sensor Network Traces**. *Adjunct Proc. 6th European Workshop on Sensor Networks (EWSN 2009), February, 2009* (Chapter 3)

Roman Lim and Matthias Woehrle and Andreas Meier and Jan Beutel. **Poster Abstract: Harvester - Energy Savings Through Synchronized Low-power Listening**. *Adjunct Proc. 6th European Workshop on Sensor Networks (EWSN 2009), February, 2009* (Chapter 2)

Matthias Woehrle and Kai Lampka and Lothar Thiele. **Exploiting timed automata for conformance testing of power measurements**. *7th Int'l Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2009), September, 2009.* (Chapter 5)

Jan Beutel and Roman Lim and Andreas Meier and Lothar Thiele and Christoph Walser and Matthias Woehrle and Mustafa Yuecel. **Poster Abstract: The FlockLab Testbed Architecture**. *Proc. 7th ACM Conference on Embedded Networked Sensor Systems (SenSys 2009), November 2009* (Chapter 7)

Adriaan de Jong and Matthias Woehrle and Koen Langendoen. **MoMi - Model-Based Diagnosis Middleware for Sensor Networks**. *Proc. 4th Int'l Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks (MidSens 2009), December 2009.* (Chapter 7)

Matthias Woehrle and Christian Plessl and Lothar Thiele. **Rupeas: Ruby Powered Event Analysis DSL** *Proc. 7th Int'l Conference on Networked Sensing Systems (INSS 2010), June 2010* (Chapter 3)

Matthias Woehrle and Andreas Meier and Koen Langendoen. **On the Potential of Software Rejuvenation for Long-Running Sensor Network Deployments**. *Proc. 1st Int'l Workshop on Software Engineering for Sensor Network Applications (SESENA 2010), May 2010* (Chapter 7)

Matthias Woehrle and Kai Lampka and Lothar Thiele. **Conformance Testing for Cyber-Physical Systems**. *In submission, April, 2010* (Chapter 5)

The following list summarizes the publications that were written during the PhD studies, yet are not part of this thesis.

Jan Beutel and Stephan Gruber and Andreas Hasler and Roman Lim and Andreas Meier and Christian Plessl and Igor Talzi and Lothar Thiele and Christian Tschudin and Matthias Woehrle and Mustafa Yuecel. **PermaDAQ: A scientific instrument for precision sensing and data recovery in environmental extremes**. *Proc. 8th ACM/IEEE Int'l Conference on Information Processing in Sensor Networks (IPSN 2009), April 2009*

Jan Beutel and Stephan Gruber and Andreas Hasler and Roman Lim and Andreas Meier and Christian Plessl and Igor Talzi and Lothar Thiele and Christian Tschudin and Matthias Woehrle and Mustafa Yuecel. **Demo Abstract: Operating a Sensor Network at 3500m Above Sea Level** *Proc. 8th ACM/IEEE Int'l Conference on Information Processing in Sensor Networks (IPSN 2009), April 2009*

Andreas Meier and Matthias Woehrle and Mischa Weise and Jan Beutel and Lothar Thiele. **NoSE: Efficient Maintenance and Initialization of Wireless Sensor Networks.**. *Proc. IEEE Communications Society Conference on Sensor, Mesh, and Ad Hoc Communications and Networks (SECON 2009), June 2009*.

Jan Beutel and Kay Roemer and Matthias Woehrle and Matthias Ringwald. **Deployment Techniques for Sensor Networks**. *Chapter in Sensor Networks - Where Theory Meets Practice, 2010*

Marco Zimmerling and Federico Ferrari and Matthias Woehrle and Lothar Thiele. **Poster Abstract: Exploiting Protocol Models for Generating Feasible Communication Stack Configurations**. *Proc. ACM/IEEE Int'l Conference on Information Processing in Sensor Networks (IPSN 2010), April 2010*

Andreas Meier and Matthias Woehrle and Marco Zimmerling and Lothar Thiele. **Zerocal: Automatic MAC Protocol Calibration**. *Proc. 6th IEEE Int'l Conference on Distributed Computing in Sensor Systems (DCOSS 2010), June 2010*

Venkatraman Iyer and Matthias Woehrle and Koen Langendoen. **Chamaeleon: Exploiting Multiple Channels to Mitigate Interference.** *Proc. 7th Int'l Conference on Networked Sensing Systems (INSS 2010), June 2010*