Towards a Test Framework for Networked Embedded Systems

Jonas Fonseca <fonseca@diku.dk>
Department of Computer Science
University of Copenhagen

March 24, 2009

Abstract

English

This thesis argues that current testing techniques for networked embedded systems are inadequate at capturing the nature and behavior of pervasive applications. There is a need for new approaches, which facilitate decomposition of these systems into testable behavioral patterns. Key challenges and work flows related with testing networked embedded systems are identified and a candidate fault model is derived. Based on these findings, a design of a test framework inspired by the aspect-oriented programming paradigm is proposed. Finally, a partial implementation of the test framework is developed and evaluated.

Dansk

Dette speciale argumentere for at eksisterende teknikker til test af forbundne indlejrede systemer ikke gør det muligt at udtrykke den basale natur og adfærd i altomsiggribende programmer. Der er brug for nye tilgange, der kan hjælpe med at nedbryde disse systemer til testbare adfærdsmønstre. Væsentlige udfordringer og arbejdsgange relateret til forbundne indlejrede systemer identificeres og en mulig fejlmodel udledes. Baseret på disse fund, foreslås et design af et testsystem inspireret af det aspect-orienterede programmerings paradigme. Endeligt, udvikles og evalueres en delvis implementation af testsystemet.

Estelle, je t'aime

Acknowledgments

The initial work on the Re·Mote Testbed Framework was done by Jan Flora and Esben Zeuthen at Copenhagen University's Department of Computer Science under the supervision of Philippe Bonnet. The Re·Mote Testbed Framework has been developed in the context of Embedded WiSeNts and has further been developed in the context of the CRUISE IST.

I would like to thank Philippe Bonnet, for his assistance and input.

Thanks to Rostislav Špinar for the collaboration and sharing of ideas.

Finally, my warmest thoughts goes to my family and friends for encouraging me.

Le silence éternel de ces espaces infinis m'effraie. –Blaise Pascal (Pensées)

Contents

List of Figures

Chapter 1

Introduction

A key objective characterizing networked embedded systems is that they disappear into the environment and work autonomously without human involvement[16]. Herein lies a big challenge, in terms of meeting the important need for testing the systems with respect to correctness, performance, and other parameters. This thesis argues that to address this, it is necessary to rethinking parts of the tool chain to provide researchers and developers with a better understanding of networked embedded systems. The main hypothesis is that current testing practices are inadequate at capturing the nature and behavior of pervasive applications and that new approaches, which more actively embrace testing techniques that facilitate aggregation and allow to decompose application flows into behavioral patterns, must be explored.

1.1 Context

To elaborate on the field of testing, networked embedded systems, and testbeds, this section provides an overview of the context in which this thesis takes place.

1.1.1 Testing as a Software Development Practice

Developing software is a complex task that requires careful planning and thoughtful management to be successful. This has increased focus on the importance of choosing a development methodology to leverage and help guide and drive the process. Regardless of these more political concerns of the development process, software development remains a very iterative and experimental process where decisions must be taken, while the system

as a whole is not clearly understood. This results in the need to constantly revisit previous steps or phases of the process, in order to reevaluate them and revise decisions. Testing is one of the most straightforward and effective methods in which to observe and measure the software, because it provides a metric on which to evaluate correctness, adherence to standards, performance, etc. against expected criteria.

Testing has received a lot of attention due to Extreme Programming[5] and the agile development movement, which have advocated for integrating testing practices into the very core of the software development process. This has multiple benefits, first the testability of software is increased by the use of patterns, such as dependency injection, that facilitates separate testing of modules and subsystems. Second, it ensures not only a robust system, but a more modular and reusable system[38]. Another important property of testing is that it serves as a form of specification of the system under test[39].

The most important goals of testing is to build confidence[5][21][39]. To quote David Gries[21]: "We should run test cases not to look for bugs, but to increase our confidence in a program we are quite sure is correct; finding an error should be the exception rather than the rule." Furthermore, this confidence is incremental in that if there is no confidence in the smaller parts, then there is none in the whole[39]. Over time when the software evolves and changes its shape, this helps to keep track and control the changes by locating regressions. Furthermore, from the perspective of economics, problems found early are easier and cheaper to fix. Consequently, effective testing keeps software alive longer[5] by reducing the cost of developing and, maybe even more important, maintaining software over its lifetime[39]. In short, the investment in testing throughout the development process pays off in many of the later stages.

1.1.2 Networked Embedded Systems

Over the last decade, networked embedded systems have emerged as a new and promising concept. Driven by the vision of pervasive computing, it is taking part in shaping the future of computing by breaking down the barriers of data acquisition and distribution. With the advance of increasingly powerful hardware platforms and standards, networked embedded systems are today being applied in areas ranging from cross-continental tracking systems to personal health care. As networked embedded systems become more widespread and mainstream, failure will become more than just an expensive inconvenience[7]. While it can be argued that it is impossible to thoroughly test a system of any real size, this clearly underlines the demand for catching defects as early as possible to not only lower the cost

of developing systems, but prevent fatal errors.

One of the big challenges with developing networked embedded systems is that they are inherently hard to test due to their scale, lack of reliability, and limitation in terms of hardware and communication. Furthermore, an essential aspect of networked embedded systems is that they are physically coupled to the environment and therefore effective experimental facilities must encompass realistic applications[34]. In short, testbeds providing researchers access to realistic test environments are necessary to understand the real-world technical challenges of networked embedded systems, such as resource limitations, communication loss, and energy constraints.

This has given rise to a number of different testbed frameworks, each of which has addressed this issue and testing challenges in widely different ways. Among the priorities are support for large-scale experiments, diversity of platforms, and optimizing utilization. However, it remains that no common testbed platform nor framework for testing has emerged from these efforts. As a result, effective and systematic testing techniques for networked embedded systems are not widely available.

1.1.3 The Re·Mote Testbed Framework

At the University of Copenhagen's Department of Computer Science work has been done to develop a testbed framework called the Re·Mote Testbed Framework[44][17]. The project is an attempt to create an complete framework of loosely connected components, which can be tailored to the infrastructure and requirements of institutions. The framework is currently deployed in the DIKU Testbed[18] and in the testbed at the Centre for Adaptive Wireless Systems at Cork Institute of Technology.

In its current form, the Re·Mote Testbed Framework provides interactive use via a rich client application. Consequently, there are several limitations in the sort of work flows and experiments, it support. It is therefore in the interest of both institutions to extend the Re·Mote Testbed Framework with a test framework to make it more versatile and able to serve as a platform for more systematic testing capable of running advanced experiments.

1.2 Research Questions

As mentioned, the underlying hypothesis is that current testing techniques for networked embedded systems are immature and do not accurately capture the needs of experiments. Our initial premise is that to address this issue, it is necessary to increase the scope and

look beyond existing research efforts done in the networked embedded system community. With this broadened scope, we seek to identify novel concepts and better understand the research space encompassing the nature of networked embedded system testing. In addition, we are also interested in finding approaches for more systematic testing as well as gathering practical experience to provide insight on how to move forward.

These concerns are captured by the following project specific learning goals, which guide the work presented in this thesis:

1. Give a comprehensive analysis of the requirements and technical problems of testing (applications developed for) deeply embedded networked systems.
2. Based on theory and similar projects, evaluate methods for building a robust test scheduling framework.
3. Design and implement a test scheduling framework.
4. Evaluate different methods for dynamically configure tests with the aim of extending the set of tested features.

During the work, the interest and focus have changed as more knowledge about the specific challenges became clear. This has let us to revise and refine the goals to focus less on scheduling and more on test frameworks from a broad perspective. Consequently, to formalize the project specific learning goals and describe the problems faced in this thesis, we pose the following three research questions:

- What challenges must be faced when designing a test framework for networked embedded systems?
- How do we approach the problem of efficient and systematic testing?
- Can dynamic testing techniques be applied to extends the set of tested features?

1.3 Approach

The first task is to understand the basic concepts of testing and the context surrounding networked embedded system testing. To accomplish, we will take an exploratory approach and look beyond existing research efforts done in the networked embedded system community through a literature study of related work.

The next task is to investigate and identify key challenges and explore the topic of dynamic testing techniques. Here we have used a problem solving approach, which groups related challenges and analyses possible solutions in terms of benefits and weaknesses. Findings from both of these exercises are summarized into a set of general guidelines and recommendations.

Based on the guidelines and recommendations, a test framework is designed. Since a functional proof of the concept is important to evaluate the design and provide a platform for future research, a partial implementation is constructed based on the design. The implementation is constructed using a bottom-up approach, which focuses on providing a client-based test platform capable of running experiments.

1.4 Contributions

The work presented in this thesis contributes to the networked embedded system community in several ways. The key contributions are:

- An overview and analysis of the challenges associated with networked embedded system testing.
- Lay out the research space for future work by identifying key areas for future work.
- A candidate fault model, which identifies common sources of faults in networked embedded systems.
- An exploration of methods to dynamically increase test coverage and isolation of faults.
- A design and implementation of a platform that will help conduct the outlined research.

1.5 Audience

Students of computer science and people interested in testing of distributed systems will find this document rich in details and discussions within this field. Additionally, embedded system developers will be interested in the discussion of the design of the test framework.

It is assumed that the reader of the document is familiar with the basics of computer science and understands the fundamentals of distributed systems. No knowledge of testing and networked embedded systems is required.

1.6 Document Structure

This document is structured as follows. Chapter 2 will first provide more background information, after which Chapter 3 takes a closer look at testing in the context of networked embedded systems. This is followed by an exploration of dynamic testing techniques in Chapter 4. The design of the test framework is presented in Chapter 5. Chapter 6 gives an overview of the implementation of the proposed system. The implementation is tested and evaluated in Chapter 7, followed by a conclusion in Chapter 8.

Chapter 2

Background

This chapter provides background on networked embedded systems presenting the specific challenges they face. Furthermore, is gives a brief overview of related work.

2.1 Networked Embedded Systems

Networked embedded systems have emerged as part of the movement behind creating a platform for pervasive computing. The goal is to instrument the physical world with pervasive networks of small embedded systems or *motes*, which are able to sense, process and interact with their surrounding environment. This poses a unique set of challenges that must be overcome.

2.1.1 Application Areas

To better understand these characteristics and challenges, let us first look at some of the application areas, where network embedded systems have been used. From the begining, inter-disciplinary scientific projects have developed and deployed these systems, because they enable close yet conspicuous observation of physical phenomena, especially for the class of applications concerned with habitat and environmental monitoring.

A good representative of the many applications in this domain is the project for habitat monitoring of birds on Great Duck Island[28]. The monitoring ran for a few months while the birds were nesting and had as a requirement that it should be as inconspicuous as possible in order not to disrupt behavior of the birds being studies. The deployed system featured a multi-tiered architecture with small battery powered sensor motes placed in the

underground nests of the bird, a transit network of more powerful motes over ground, and a base station for collecting the sensor data. The project showed that networked embedded systems, while still in the research stage, had a lot of promise in shedding new lights in areas where it was previously difficult to collect information.

Other examples include applications for tracking long-term animal migration in Africa[45] and efficient parking management in urban areas[8]. Networked embedded systems have also been applied commercially to areas, such as tracking warehouse inventory and providing detailed information related to logistics. A great potential is also in the area of reducing energy consumption, where networked embedded systems can provide insights into the efficiency of manufacturing processes[11] and help to reduce waste.

2.1.2 Common Characteristics and System Challenges

While each application area has its own requirements and restrictions, the systems all share some of the same characteristics. They are all deeply distributed systems consisting of resource constraint motes, which have to operate in extremely dynamic environments. The dominating factors are the requirements to the form-factor, the cost of hardware, and the life-time of the application.

The deployed motes generally consist of small, low-cost, energy-efficient hardware platforms, which are then equipped with components for acquiring and storing data depending on the requirements of the application. By scaling down each mote, dense instrumentation of phenomena under observation and the creation of systems of immense scale is made possible. This poses the challenge of creating autonomous systems which can operate with limited human access.

While some systems rely on harvesting energy[45], they are most commonly battery powered. As a result, dealing with the energy constraints poses the most dominating challenges in the design of the overall system. The reduced energy budget requires that the motes use low duty-cycling, where they power down unused hardware and spend most of their time with the main processing unit in ultra-low energy sleep-mode.

The single most expensive operation in these networks is communication, which is provided via short-distance radios. To cover larger areas, the systems rely on forming ad-hoc networks and use multi-hop and location-based routing schemes to efficiently propagated information to its destination. To minimize communication, in-network processing and topology-aware aggregation must be applied. Furthermore, the limited uptime of motes combined with the unreliable radio medium results in high-latency of data propagation,

which means that communication must be fault- and delay-tolerant.

Using strategies such as "sense and send" and "store and forward", applications are highly driven by interactions with the environment and often have soft real-time requirements. This means they have to achieve a high level of concurrency, especially on platforms relying on cooperative scheduling. Lastly, the immense scale and limited access, means that it is difficult to retask and updating application once they have been deployed.

2.1.3 Programming Models and TinyOS

In terms of software, applications are deeply tied to the hardware. Furthermore, while a high diversity of platforms offer a lot of choice in terms of customizing the hardware, the software/hardware boundary is often not well-defined and there is a lack of general abstractions. The main reason is that resource constraints force the use of optimizations and specialization in favor of abstractions in order to tailor applications to run efficiently. Reevaluation of algorithmic techniques and use of approximations are often necessary to accommodate for uncertainty and reduce the overall execution foot-prints.

To overcome some of these challenges, work on developing a more appropriate programming model has been undertaken. One of the most successful programming models for networked embedded systems is the model developed for TinyOS[22], a small operating system-like layer. To support this model the language nesC[1] has been developed, as a dialect of C with a small set of domain specific constructs.

The most important language construct is the support for decomposing systems into components, which expose functionality via interfaces. Applications are then composed by wiring components together, whereby it becomes easy to replace components and thus customize the system to the individual platforms. Furthermore, by making it possible to define components as thin layers, which wraps hardware components, the problem of defining a software/hardware boundary is overcome.

The TinyOS programming model also provides an abstraction in the form of a simple two-level execution model to support the highly event-driven nature of applications. Using cooperative scheduling, long running tasks, which need to process data from the network or sensors, can be posted to run in FIFO order. Events in the form of interrupts from hardware, preempt the execution of tasks and allow management of hardware.

The force behind nesC is that it allows whole-program compilation, which brings two major advantages. First, applications are preprocessed and assembled into one single code

[1] networked embedded system C

artifact using compile-time bindings based on the component wiring. This significantly reduces the cost of the language abstractions and allows very aggressive optimizations. Second, using whole-program compilation the compiler is able to do static checking of the complete control-flow of the program with knowledge of the execution context for the different parts of the code. This helps to reduce defects by detecting problems, such as race conditions caused by variables being accessed non-atomically from both normal context and interrupt context.

2.2 Related Work

Inspiration for the work presented in this thesis has come both from research in the field of networked embedded systems and sensor networks, but also from the research of distributed system in general. In the following a brief overview of related work is presented.

MoteLab[40] was one of the first testbeds developed for networked embedded systems. Using a networked backchannel of permanently deployed and powered motes connected to a central server, remote reprogramming and monitoring of the motes are available through a web interface where users can schedule jobs and collect data from test runs. In addition, users can access nodes in real-time to use custom programs for monitoring and injecting data into the running application.

One of the founding ideas of MoteLab is the use of active messages, which can be compared to an extensible packet format. By defining parsers or packet decomposers, messages received from the serial consoles of motes can be logged into a database together with a timestamp, allowing the tester to use them to later recapture and reason about the states of the individual motes. While feature-rich, the MoteLab design choices put significant limitations on what kind of experiments it can handle[6] and the important aspect of resource management.

Some of these limitations are addresses in Mirage[9], a microeconomic resource allocation scheme for testbeds. Chun et al. found that flexible sharing of a sensornet testbed in both space and time is necessary and argue for an auction-based reservation system, where users compete for resources, to address the problem of resource contention. By placing bids, users contribute to maximizing the testbed usage, which makes the system more flexible and better at coping with low and high resource demands compared to a strictly quota-based system with fixed-valued time slots. However, they also note that such a system gives users an incentive to try to game the system during heavy demand, and suggest

that strategy-proof or hard-to-manipulate auction mechanisms needs to be considered to avoid pathologic gaming of the system.

Abstract resource specification is another important method to extend the flexibility and usage maximization of a testbed. Mirage uses a combinatorial approach, where requests are made for collections of motes with certain characteristics, such as platform and topology, opposed to specific physical nodes. This more accurately allows users to express their preferences, and enables the system to find the best mapping and schedule multiple specifications simultaneously.

A more elaborate system for mapping of testbed resources has been developed for Emulab[36]. It takes the approach of using virtual equivalent classes and simulated annealing to reduce and efficiently solve the mapping problem. Ricci et al. also address the problem of minimizing inter-experiment effects by considering node localization and using load-balancing, something that is not addressed in the work on testbeds for networked embedded systems.

Outside of the research community, testing of distributed systems has been addressed by Selenium[23], a system for testing web applications. The basic idea behind the test framework in Selenium is to use existing web-based technologies to make it possible to run across multiple platforms. First, a remote control server is used for orchestrate the tests by launching browser instances and acting as a proxy between the browser and the web application running on the server. By using a web proxy, it is possible to inspect both what data the client requests and what data the server sends. Second, the browser's own capability of executing JavaScript code is used for embedding a test automation engine on the client side. This method gives full access to inspecting the document content and the browser state as well as controlling the browser's behavior, such as clicking on links. Furthermore, Selenium provides different means to create tests; one is the notion of test drivers, which are external programs, which can control the whole test process. This way Selenium can be integrated into whatever test frameworks project are using. While Selenium's architecture is very powerful and easy to use, one downside is that tests are brittle and may easily break[24]. Another problem is that the framework does not scale well when the size of the test suite increases, because the remote control mechanism is slow and has limitations in terms of how much parallelization is possible.

Chapter 3

Testing Networked Embedded Systems

In this chapter, we will explore testing of networked embedded system in terms of several major topics. First, the key challenges related to testing of embedded and networked systems are presented using a problem solving approach. This is followed up by looking at some of the work flows specifically involved in the use of testbeds. We will then define a fault model, which recognizes the typical source of faults in networked embedded systems. Finally, we summarize the findings by providing guidelines and recommendations that may be used in the design of a test framework.

3.1 Key Challenges

Networked embedded systems are characterized by their limited hardware capabilities. These limitations makes it difficult or even impossible to test certain features of the system. It is therefore necessary to clearly identify these challenges to design a robust test framework for networked embedded systems.

3.1.1 Non-deterministic Hardware

One of the biggest challenges is dealing with non-deterministic hardware. The platforms used in networked embedded systems are cheap, which leads to behaviors that are unpredictable. In other words, it is necessary to consider how to deal with the irregularities caused by non-deterministic hardware to avoid that they disrupt experiments.

If we look at the causes, some are due to the hardware being broken. Hardware quirks can also cause problems that may be perceived as if they were caused by non-deterministic

hardware. An example from the motes in DIKU Testbed is the radio's inability to send messages having an odd length. Other causes can be attributed to interference from external source. Examples include sensors sensitive to static electricity.

It is very important to reduce the impact of non-deterministic hardware, as it can be very disruptive from the user's perspective. More importantly it can seriously affect the ability to reliably reproduce experiments, whereby the fundamental property of the test framework violated. To avoid these problems, the health of the testbed needs to be monitored continuously. The monitoring can involve tasks, such as running small applications, which do on-board self-test on each mote. In case of problems, the affected motes must be excluded from any future experiments.

3.1.2 Testable Units

It is often desirable to confine tests to focus on a specific part of the system. This allows testing to be performed incrementally in a bottom-up fashion, where test of small self-contained components are first performed and then gradually extended to more complete parts of the system. This sort of incremental testing going from self-contained components to complete systems helps to limit the scope and ease locating the origin of errors. To achieve this form of granularity, the test usually needs to emulate surrounding components with which the component under test. This requires that the system is modular and has been designed with testability in mind.

The dependency on cross-component optimization and integration make it challenging to find well-defined boundaries between the components making up the system. The main problem is that the resource constraints may not always permit programs to be structured into modular and testable components. As a result, it may be necessary to test bigger units or create a lot of extra stubs to replace core components with the result of completely changing the system behavior. This is to some extends less of a problem when using TinyOS and nesC, which allow replaceable components to be created. However, while components can easily be replaced certain components cross-cut the component graph and are thus difficult to replace.

In order for testing to help locate defects, tests need to be able to target specific parts or units of the system. While existing techniques address this problem through the use of stubs, we still see a potential in more thoroughly investigating what kind of structure and granularity is desirable for the specific case of networked embedded systems.

3.1.3 Observing the System Under Test

Another big challenges when testing embedded systems is the limitations in the ability to observe the state of the system under test. Tracking the state of a running system is often only possible via debuggers, and only through external devices, such as integrated circuit emulators (ICE). These devices can be very expensive, raising the cost considerable if a complete system of multiple motes needs to be equipped. Furthermore, ICEs can be slow and unpredictable due to the way they emulate certain parts of the underlying hardware.

Tracing using probes is a more straightforward and proven method of observing embedded systems. The probes can either be in the form of software probes, which increment a counter or record information in a buffer, or it can be in the form of an oscilloscope[19] or a similar measuring tool, which is attached to a part of the electrical circuit. Using the latter method the test code can flip the power of for example a pin on an unused microprocessor to signal a change in the state.

While tracing offers one of the most rigorous methods of tracking the state of an embedded systems, it can potentially generate a lot of test output. In some circumstances, such as long-lived experiments, the data amount may exceed the capacity to process or store on-board during execution and have to be transferred to an external system via the mote's serial console for post-mortem analysis. Limitations of the throughput and reliability of the serial console, may require that data is compressed and sent using check sums.

3.1.4 Detecting Failures

A similar challenge lies in using the observations of the system under test to determine if the system behaves correctly. This leads us to the question: how do we detect failures? Or more interesting, how to we detect correct behavior? To answer these questions, we must first understand how failures occur.

As already mentioned, system engineering is a highly iterative process. Consequently, decisions and choices are made before the whole system is clearly understood and evolve over the course of a project. These decisions and choices ends up in the resulting system as assumptions and invariants, some of which are faulty from conception or will become faulty because external assumptions changes. These assumptions are a potential defect; the defect may and may not cause a failure, depending on whether or not it will be stimulated. If a defect is stimulated under the right circumstances, it can cause an infected state in the system. Once an infected state is present and it is not corrected or otherwise masked

by subsequent events, it can propagate and spread to other parts of the system. In a distributed system, this may lead to the infection spreading across the boundary of a single system. At some point the infection will have become so severe that it will cause a failure, whereby the defect will become visible to the surroundings[42].

From the above, it should be clear that there is not always a clear link between the source of a fault and how the fault manifests itself. A potential failure may stay hidden for a long time before it can finally be detected. In systems with a very low duty-cycle, it may take time in the amount of days to get the system into a state where the failure shows itself. Furthermore, how a failure presents itself in the output of the experiment might not always be clear. In this sense, for certain failures, you have to know what you are looking for in order to be able to detect them. To give an example, a dead lock in an application using a watch dog timer might only be visible by the fact that motes will restart periodically.

The most straightforward approach to detect failures is by looking for anomalies. This can be done by comparing the traces and output from the experiment with the output of an expected behavior. The approach can be further extended by creating an abstract representation of the expected output either by looking at multiple runs or by explicitly creating such an abstracting. It is still unclear how such a representation is best described and whether it is feasible to automatically derived it from test output. It remains, however, that effective testing depends on the ability to detect failures as early as possible and as reliably as possible. To accomplish this good test evaluation techniques needs to be developed.

3.1.5 Instrumentation and Use of Artifacts

Certain properties of the system under test are hard to infer only from information gathered via methods, such as tracing. It may therefore be necessary to instrument the system with software components, which can collect the required information. There are several areas where this can be useful during the development phase.

The biggest potential is in terms of gathering runtime artifacts. Here instrumentation can be used to get different types of statistics and detailed information about the system under test. This can give valuable information about the performance of the system, e.g. duty-cycling. It can also play an important role in increasing test efficiency by assisting developers in determining the adequacy and coverage of test suites. We have already mentioned the use of static analysis, which capitalizes on compile-time artifacts in the

form of call graphs to infer the control flow and data flow of the program whereby potential software defects can be identified. It is interesting to explore if the use of compile-time artifacts can be combined with code generators to automatically create the components for instrumentation. This can reduce the amount of manual work related with gathering runtime artifacts with a satisfying level of detail.

It is important, to use instrumentation carefully. The main reason is that timing is crucial for embedded system and applications are event driven and may have soft real-time requirements. If the method is not used wisely, instrumentation of the system can potentially have an effect on the behavior of the system under test. This can lead to surprising differences in behavior, when the instrumentation is disabled affecting the ability to reliably reproduce experiments.

While there are problems with using instrumentation of embedded systems, there are still valid uses to explore. For example, it can enable fine-grained control of experiments because on-mote components have access to the input and output of the system under test. One possible use is for injecting sensor data into an experiment using a predefined model of a phenomenon, which can make experiments both more realistic and reproducible.

3.1.6 Reliable Reproduction of Experiments

To reproduce an experiment several stages needs to be orchestrated.

Preamble where the system under test is brought into the initial state required by the test case.

Body where the actual test is performed.

Postamble where the system state is restored with the main objective to bring the system into a state where another test can run.

During this execution, timing is challenging, since multiple motes in the system must first be brought to a desirable state during the preamble. Then a series of events needs to be triggered at precise time offsets and reliably across the system to force the behavior or state, which must be tested, during the body. This poses several challenges in terms of the latency for the underlying testbed infrastructure, especially if the experiment needs to be executed remotely via the network.

3.1.7 Tracking Communication

The ability to reason about networked embedded systems depends highly on the potential to track the communication. In terms of networked embedded systems, the research community initially sought to start afresh by not basing their work on existing network abstractions. The main argument was that the existing networking approaches was not applicable to these new constraint systems and it was necessary to open up the research space to new ideas[26]. This has lead to a lot of exploration of how to compose a network stack for networked embedded systems. In other words, the communication in terms of protocols and messages is highly application specific and thus not easy to handle generically in a test framework.

The first challenge is to decompose communication by parsing the fields in the messages used by the system under test. Next challenge is the ability to interpret the meaning of each messages, such as whether they contain control data. By accomplishing this, the test framework will be able to move beyond passive listening to inject messages into the system under test and stimulate it.

One possibility employed in MoteLab to solve the first challenge is to explicitly associate message parsers with a test. Hereby, messages can be decomposed by the test framework and inserted with fields into a database for later retrieval. Furthermore, nesC has support for generating such parsers from annotated source code. With the advance of more consistent use of network standards and protocols in networked embedded systems, such as IPv6LowPAN and IEEE 802.15.4, some of these challenges will no longer need to be considered.

The communication strategies in networked embedded systems also poses a challenge. Timeliness of tests is of great importance for the overall usability of the test framework. The conservative energy budget and resulting limitations of radio usage means that strategies such as delay-tolerant networking are used, which results in experiments requiring a long time to run taking up valuable resource. A solution is to investigate if it is possible to speed up test without affecting the validity of the results. Imagine a coordination protocol, where motes wake up every 30 seconds and synchronizes with a master. Would it be possible to reduce this wake up period to only 15 seconds without breaking the general assumptions of the system as a whole? Similarly, in a scenario where events are being replayed, could the running time of this test preamble be reduced by speeding up the replaying of events or even leaving out certain unrelated events?

3.1.8 Application and Platform Diversity

As the research field has progressed, many of the technical challenges related to system architecture and networking layer have been addressed. This has shifted focus towards some of the more inter-disciplinary scientific challenges. Herein lies a great opportunity but also an great challenge for testbeds. On one hand, there is a potential to facilitate the requirements of realistic application, however, this also comes at the cost of affecting the diversity of applications that can be supported. In other words, there is a trade-off when choosing which mote platforms and sensor boards to deploy. This is one of the reasons why several different kinds of testbeds targeting different applications are needed[34]; some mobile and customizable to specific application needs, others larger and more permanent for experimenting with sensor networking primitives at a significant scale.

Another fundamental problems is platform diversity. Many different platforms exist, each of which has specific interesting properties that make it suitable for a specific application or research area. For example, to facilitate research in emerging technologies the testbed must offer cutting edge platforms. Furthermore, for each platform different alternatives may also exist in terms of what sensors they provide. Given the multiples of platforms and sensor configurations, it is unlikely that any testbed will be able to provide for the demands for a specific platform or a specific sensor.

For dealing with these issues, an interesting solution, which has been explored in the context of PlanetLab[32][31], is federation of testbeds. While it raises questions about how resources are exchanged, there is much to be gained in terms of greater deal of cooperation and sharing of experiences among testbeds.

3.1.9 Topologies

In order to facilitate a wide spectrum of scenarios, the test framework must facilitate the configuration of many different topologies. For example, an experiment might need to test against a specific category of topology, such as star topology, using different levels of densities ranging from a very sparse network to a more strongly connected network. One of the challenges related with topology and configuration is that writing a test, which configures a specific topology is difficult, and, more important, gets repetitive and tedious when the application needs to be testing against different topologies. Furthermore, by having to manually configure a topology, knowledge of the testbed infrastructure is require, which, due to its potentially dynamic nature, can cause experiments to become brittle

The simplest solution is to use on-mote components to configure the topology by simply filtering mote messages. However, this may require recompilation of the mote program. Another solution is to configuring the topology by picking motes to include and motes to exclude given that the testbed is big enough. However, for densely deployed testbeds this is not a feasible strategy, because it will result in a very small topology area.

Another concern related with topology is utilization of testbed resources. Certain experiments might require a topology area covering the whole testbed, but without the need for using all motes in the testbed. For this case it is worth exploring if utilization can be increased by separating the communication of experiment to different radio channels. While this type of "multi-tasking" is desirable it requires the cooperation of experiments, since confinement to a specific radio channel is hard to enforce.

3.1.10 Dynamic Environments and Infrastructure

Similarly to non-deterministic hardware, there is also a big challenge of making environmental effects have as little impact as possible on the system under test. For example, noise from surrounding wireless networks can interfere with an experiment and affect it in multiple ways. While this may have positive implications in that it can uncover new defects in the system under test, it will nonetheless end up making it impossible to later replay the original scenario and reproduce the defect and thus verify a possible fix. Consequently, it may be necessary to not only monitor the experiment itself but also the environment in order to later be able to reason about a test run. It may also be appropriate to make it possible to adjust test cases at runtime to accommodate for environmental interferences.

Simulation provides a solution for this and many of the other challenges mentioned above. Among the advantages is the ability to test very large networks potentially only limited by the available computing power and the accuracy of the underlying models used in the simulation. It also has the benefit of given more control over the experiments, for example by trading accuracy for efficiency and timeliness or by testing against different radio propagation models. Finally, simulation also give the unique opportunity of experimenting with sensors for which hardware does not yet exist[20].

While simulation can be used to evaluate algorithms and understand the causes of behavior observed in the real world, it should, however, not be used for absolute evaluation without also considering whether the scenario being modeled has basis in the real world[27]. Furthermore, compared to simulation, only a testbed using real hardware can provide a solid and thorough understand of the real-world technical challenges of networked embed-

ded systems, such as resource limitations, communication loss, and energy constraints.

An compromise is to use a hybrid solution in which experiments run on both physical and simulated motes[15], whereby experiments can be scaled appropriately, while still providing realistic output from the physical motes. This depends on the ability to exchange parameters between the physical and simulated world. It remains that the dynamic and potentially non-deterministic environment are a source of problems, when it comes to ensuring the reliable execution of experiments.

3.2 Work Flows

With the challenges in mind, we will next look at different work flows associated with the use of testbeds.

3.2.1 Work Flow for Experiments

The main goal of testbeds is to provide a ready to use and automated facility and overcome many of the time consuming tasks associated with running experiments on real motes. The main tasks that helps to accelerate the execution of experiments is the simplified data collection. The most important eliminated tasks are manually reprogramming of motes, deployment of motes, and collecting of motes after the experiment. This leaves the following basic work flow for experiments:

1. Create an experiment by selecting motes and assigning binaries to run on them.

2. Schedule the experiment to run.

3. Collect data from the testbed after experiment has run and analyze it.

There are a lot of benefits with this model in that once an experiment has been created, it can be rerun multiple times, whereby the initial overhead of creating and configuring it is amortized. Automation can also help to increase the general utilization of the testbed, because the resource usage of experiments can be estimated in advance, which allows more effective scheduling and even the possibility to run multiple experiments in parallel.

3.2.2 Interactive Usage

One of the problems with the basic work flow is that it requires that many decisions are taken upfront. This makes the work flow less ideal for use during the initial stages of a development process, where there is more demand for experimenting and the development cycle of programming and running the program is quicker. While some testbeds allow semi-automated experiments where users can control a running experiment, this still suggests that there is also a need for supporting interactive work flows, because it allows to incrementally configure and derive the desired scenario. Interactive use is also preferable when learning an API, such as when students need to familiarize themselves with a radio stack at the start of a course.

The basic work flow for interactive usage is very similar to the one above, except for a few extra steps. However, some steps can be skipped for subsequent experiments during the same session.

1. Select and take control over motes.
2. Program motes by assigning a binary to them.
3. Instrument data collection.
4. Run the experiment by starting and stopping motes in the desired order.
5. Collect and analyze data.

One problem with interactive usage is that it can lead to usage patterns that are not optimal. Experiences from the DIKU Testbed shows that users keep themselves logged in even for small tests potentially depleting mote resources. There is of course also a challenge in offering both interactive and non-interactive experiments in that the sharing of testbed resources is not as well-defined. One possible solution to offer mixed work flows is to require resource reservations for all experiments.

3.2.3 Use Cases

To further guide the design choices, it is useful to also establish some general use cases for testbeds. We will focus on two use cases in the following: one where a researcher is experimenting with finding an efficient routing method and another where a student is working on an assignment for a course in networked embedded systems.

The Student

For the student, it is the first experience in using a testbed. It requires some time to figure out how to get the first experiment working, but after this initial obstacle the student spends some more time to get familiar with using the testbed by gradually extending the experiment. After this first experience, the student has good idea about how to integrate the use of the testbed into the work on the assignment and has, furthermore, developed a fairly advanced experiment, which can serve as a good basis for future work.

The Researcher

The researcher has a lot of experience with using the testbed and also knows some of the problems and limitations from running previous experiments on the testbed. Since most of the routing components are developed the project is entering a phase, where the experiments have a larger scale. However, the researcher is still interested in also running smaller experiments when tuning specific parts of the behavior because the large scale experiments take a longer time to run and is usually run automatically during the night.

The chosen use cases show that the test framework must be versatile to support both types of users. This clearly underlines that there is a large gap in terms of being able to both get new users started and support advanced users needs.

3.2.4 Supporting Multiple Work Flows

Rather than trying to bridge the gap between the requirements of different work flows, we believe that the test framework should instead strive towards exposing different layers of functionality, which enable users to chose the level of control they want. We see several stages of an experiment, which could be improved by such functionality. First and foremost, the ability to control the motes themselves when creating experiments. While again different work flows has different requirements, even simple experiments need some sort of control, for example to program and start a mote. Second, the ability to store data collected during an experiments in a database can help to make the data more accessible and uniform. Finally, post-mortem analysis of experiments, for example when testing regressions or trying to isolate a defect.

Furthermore, we believe it is unlikely that any single tool will suit all users, and instead propose that the test framework is made extensible. This is also a recognition of our belief that the success of the test framework very much depends on its ability to be integrated into the tool chain developers are already using and familiar with. This will also enable the testbed to offer its presence in many different ways, such as via both web and desktop applications. Finally, this will allow future work to rethink how users can interact with the testbed.

3.2.5 Running Non-interactive Experiments

Certain work flows require that non-interactive experiment can be run a testbed server, because it allows better utilization of testbed resources and can help users to manage long-running experiments. However, it also opens up for many new challenges. Most important are the many different security considerations that must be addressed. Because experiments can potentially execute arbitrary control code, they must run in a sandbox. Similarly, several experiments may run simultaneously, which can lead to degradation of access to server resources. To accommodate this, the system should employ protecting measures and actively monitor the experiments that are running so it can intervene if their resource usage is deemed harmful for the rest of the system.

A similar challenge is related with how the output generated by experiments is handled. When running locally the output can be stored in files or logged into a database that the user controls. However, when running non-interactively, the experiment needs to abide by the rules governing the server, such as disk quota and other storage policies. Finally, a service must enable the user to fetch and manage the output from past experiments.

Another issue is with how to deal with the aspect of unpredictability in the execution of experiments. Certain failures may require human intervention, for example in the event that a crucial part of the experiment, such as a gateway mote, is not able to run as expected, or the user accounts' disk quota is depleted. Where the user can simply decide to restart an interactive experiment, it is less straightforward how certain failure scenarios are best handled for experiments running on a server.

Finally, the test framework needs to provide a service for submitting and managing experiments. This service must consider whether experiments are stored on the server, so they can easily be rescheduled to run, and whether they are regularly deleted. Together with the above challenges, this clearly shows that there are many policy-related issues with non-interactive experiments, which must be addressed.

3.3 Fault Models

We will now focus on analysing how to move towards a more systematic testing approach. Given the unique challenges, networked embedded systems introduce new types of faults, which are characteristic for these systems. One way to identify them is by defining a fault model[4]. The aim of a fault model is to create a taxonomy of the different types of faults by identifying common errors that are likely to occur in an implementation. This way potential sources of faults are highlighted and grouped into problem areas, which can serve as guidelines during test specification.

3.3.1 The Nature of Faults and Failures

Before we can define a fault model, we need to first examine the different sources of faults and failures. From Section 3.1.4, it should be clear that there is not always a clear link between the source of a fault and how the fault manifests itself. It depends on many different circumstances for the right chain of events to occur. However, it is possible to narrow down the potential sources to basic entities and relations in the system. We identify the following four potential sources of faults in networked embedded systems:

1. **The fault resides in the hardware.** These faults occur due to the use of low-cost hardware platforms and extreme environmental conditions.

2. **The fault resides in the mote software.** This source of faults occur due to defects in the software. For example in the form of buffer overflows, when static allocations are incorrectly accessed.

3. **The fault is an emergent property created by interaction between the hardware and mote software.** Such faults occur when assumptions in software do not match those of the hardware. Example sources include race conditions and drivers, which do not account for reentrancy in interrupt code[35].

4. **The fault is an emergent property created by interactions between motes and their environment.** These faults occur due to incorrect assumptions regarding the dynamic environment.

3.3.2 A Fault Model for Networked Embedded Systems

From our findings based on the work presented above and the study of researches, we propose the following candidate fault model for networked embedded systems.

Incorrect system assumption and simplification

In order to accommodate resource constraints, applications need to simplify and make assumptions about the environment. This may be in terms of local properties, such as the ability to store sensor readings for longer periods of time. If assumptions are oversimplified or the system lacks degraded modes, which kick in in case of failure, the application as a whole can come under pressure and end up collapsing.

Failure to meet concurrency requirements

Networked embedded systems are integrated into environments of extreme dynamics. This makes it very essential that concurrency meets the soft real-time requirements of the system.

Failure to self-organize

In order for networked embedded systems to be scalable, they must be able to organize themselves. Part of this is the ability to form and continuously manage topologies that offer a level of routing quality demanded by the application. As an example, applications needs to ensure that proper measures exists, which cope with topology changes.

Incorrect expectations for quality of service

The success of the system is determined by its ability to live up to the level of quality for the services, which it is expected to provide. Specifically, a sensor-based application needs to match the capabilities of the system with the level of detail and precision with which to observe the phenomenon of interest. To accomplish this the quality requirements may employ lossy modes based on how collected data will be used.

3.3.3 Benefits of Fault Models

The main motivation behind defining a fault model is the idea that to be effective, any systematic testing strategy must be based on some notion of the type of faults that can

occur in an implementation[30]. Furthermore, a fault model can aid in developing tools and determining coverage strategies specifically tailored to the system under test. Consequently, we argue that systematic testing of networked embedded systems must be based on fault models, which reflect and account for the specific system characteristics.

We consider the above fault model a candidate, because more work is necessary to extend the model and more clearly identify common patterns. Furthermore, a definitive model should also provide criteria for how to test against each fault. We also propose a study of whether or not faults, which are a result of emergent properties, require more test effort.

3.4 Guidelines and Recommendations

To summarize the findings, we will provide a set of general guidelines and recommendations for how to design a test framework for testing of networked embedded systems.

3.4.1 Guidelines

By now it should be clear that designing and writing tests is not always easy and straightforward. There are many issues and trade-offs to consider in order to write efficient tests. For test cases and test suites we identify the following guidelines:

Help to locate the defect The granularity of the unit being tested should be small enough that locating in what part of the system the defect resides does not take valuable resources. One way to achieve this is to partition the test suite into logical parts, each of which focuses on testing a particular part of the system. Furthermore, the test suite should be arranged so that it uses incremental testing, aggregating result by starting with smaller units after which it gradually increases the scope.

Reliability and reproducible It needs to be possible to reliably reproduce test failures. This means that the test should account for all the possible different aspects of the test environment such as the architecture and platform, on which the system runs. Another important aspect is to account for specific timing requirements.

This can be especially tricky for performance and randomized tests, where part of the test environment cannot easily be controlled. In both cases, this sort of behavior needs to be documented in the test results along with information critical to later attempts to rerun the test or perform post-mortem analysis.

Coverage Coverage are important to enable effective testing. Exhaustive testing is infeasible, which means that it is necessary to select a subset of the possible test cases. The test selection criteria should try to increase the coverage of tests. Achieving this requires knowledge of the system under test in the form of run-time and source code artifacts, which can provide information about the control and data flow.

Timeliness and responsiveness For testing to act as a useful tool and something developers take serious it needs to be fast and responsive. The time to test the part of the system of interest should not be greater than a few minutes to keep ideas fresh in the minds of the developer. This is somewhat contradictory to the goal of having coverage. For this reason, it may be necessary to have several test suites for large systems, where a minimal and fast test suite covering the most basic functionality can be run in each smaller development cycle and a test suite with full coverage is run automatically and periodically on a server.

Maintainable Tests like all other software needs to be maintained as the surrounding system evolves. This suggests that as patterns emerge, tests may need refactoring and have their own set of abstractions. Furthermore, tests should avoid making assumptions about the test environment that can make them brittle.

For test framework we define the following guidelines:

Simple to write tests A test framework must make it possible for skilled as well as novice developers to write tests.

Automation Another crucial feature to lower the barrier to testing is automation. Making the testing process as automated as possible is really essential to ensure that the test suite is run frequently. Automation gets especially important when deadlines advance and the stress level rises.

Generic and versatile Being a framework, it is very important that it provides a very generic and general usable interface for writing tests in order to support many different work flows. The most important issue here has to do with allowing tests to be controlled.

A versatile test framework should also provide support for reducing the amount of work required for common idioms and testing tasks. Part of this has to do with allowing an easy way to setup and teardown test cases via preamble and postambles, which removes the need for boilerplate code.

Highlight errors and breakage Spotting problems related to errors and breakage is the main reason why testing is done, therefore this should be a top priority. This means that it should not require any manual work and that it should be both easy to see which test suite and which specific test failed. Furthermore, if possible information about the breakage in terms of how the test result differs from the expected.

3.4.2 Recommendations

A: ***The test framework must be based on a holistic approach to testing and address the issue of testing complete systems.***

While testing of units is important and helps to increase the confidence in the overall system, it can be argued that it is just a specialization of the general case, namely testing a system of motes. Furthermore, creating a unit test system, e.g. for TinyOS, is fairly straightforward.

B: ***The test framework must not rely on instrumentation of motes.***

First of all it is problematic to depend on what features are compiled into the software deployed in the testbed. Second, by requiring instrumentation, it is likely that certain results will not be realistic.

C: ***Instrumentation may be offered as an opt-in.***

For example for determining test coverage of applications or to account for limitations in both the testbed and motes.

D: ***The test framework must be able to express and execute complex experiments of considerable scale.***

The effectiveness of the tests depends on whether or not they are able to model advanced scenarios in a scalable manner.

E: ***The test framework should encourage extensibility through open APIs.***

It is necessary to acknowledge that to fully embrace new ideas it is necessary to provide developers the ability to extend the test framework and integrate it into applications and tools.

F: ***Information from software and runtime artifacts should be included into the test framework.***

There is a great need to including information from many different sources to increase test efficiency and thus the usability of testbeds. This depends on a higher degree of integration of the test framework into the overall development process.

Chapter 4

Dynamic Testing

Testing is about decomposing systems and applications into units, which behavior it is possible to reason about. This is challenging for networked embedded systems because behaviors are emerging and phenomenon cross-cut the system across multiple motes. In this respect, the problem with normal testing in that it does not scale in terms of tracking events and behaviors in a complex system. As a result, developing and maintaining test suites with sufficient coverage is a very laborious task. To approach this problem, it is interesting to evaluate different methods for dynamic testing systems by extending existing test suites and automatically generate tests. The following sections will explore this topic in terms of detective measures, such as random testing and model-based testing, and corrective measures concerned with isolating software defects.

4.1 Random Testing

For any system of considerable size there is a necessity to deal with irregularities in software. In this respect, randomized testing can be a useful method to find behavioral corner cases for which the developers have not accounted or simply check how the systems accounts for unexpected situations. We will explore random testing in the context of random event schedules.

4.1.1 Random Event Schedules

The basic concept behind random event schedules is to introduce test drivers, which generate random events or behaviors during a test, and observe how the system responds. To

give an idea of how these test drivers can extend an existing test cases, assume that a fairly generic test suite already exists. This test suite is initially run unaltered to get a reference of expected output. Next, the test suite is rerun with one or more test drivers, each of which inject some sort of randomized events. The events are created using the controls of the test framework or secondary motes that inject messages.

What kind of test drivers can be applied of course depends on the system under test and the test suite. However, we propose several options:

Message replaying This test driver will randomly pick a message or a sequence of messages and "replay" them. The replaying can be done via motes, which are not directly part of the experiment. The main objective of this test driver is to test how applications, which allow routing through multiple paths or use flooding strategies to disseminate information, handle messages arriving multiple times.

Radio noise Radio communication is not reliable, which means that strategies for handling noisy channels needs to be applied. This test driver randomly introduces noise during an experiment with the goal of disrupting radio communication. This can either be done by sending bursts of messages with random bytes or garbage messages.

Topology change As we have pointed out, networked embedded systems needs to be able to self-organize in case of failures. This test driver will randomly pick a mote and exclude it from the experiment by turning it off. Depending on the density of the used topology this can be repeated multiple times to the extend where the test driver can cause a partitioning of the network. The objective is to test resilience to topology changes in the routing layer. As an alternation, motes can randomly be added to the topology as is the case during the initial deployment of a system. This can be used for testing synchronization protocols.

4.1.2 Using Random Testing

One of the benefits of using random testing techniques is that they are relatively easy to implement and require little work and instrumentation of the code. Furthermore, they can be applied at different testing levels. As an example, it has been applied to harness testing of interrupt handling in embedded systems[35], where it was found that interrupt handling code is often a source of defects because assumptions do not account for spontaneous interrupts, for example when triggered by hardware flaws or an external electrical discharge.

When performing randomized testing, it is important to save enough information to allow the tested scenario to be replayed and reproduced, if it should later be necessary. This can be done for example by recording all events, which happens during a test, or using seeds for generating event schedules. The latter, requires that the test driver's event generator uses a deterministic algorithm.

One problem with using random testing is to account for the case that a test no longer only fails or succeeds, but gives an "unresolved" result. Using the above examples of test drivers, a test of a system may be unresolved if a mote failure results in a partitioned network or shuts down a mote, which has a special role in the experiment, such as being a gateway.

While random testing can effectively elicit defects, which are otherwise hard to find, it potentially lacks in terms of efficiency, because searching for defects by randomly altering test cases requires a lot of test executions. Some improvement can be introduced by using heuristics to make more directed random testing, for example by defining rules that can be used to limit, which events schedules are considered. However, as we shall see, other and more efficient techniques exists for dynamically extending tests.

4.2 Model-Based Testing

A more directed approach to testing is to use model-based testing techniques for generating or extending test suites. In some sense, all testing is necessarily model-based in that any test case is based on the testers mental model of the system. Furthermore, we believe that modeling is an integral part of developing networked embedded systems, and something that a developer is likely to do anyway as part of the specification or design process, be it explicit or implicit. From this perspective, model-based testing is simply a method to make these activities more formal.

The general idea is to use detailed models, which accurately describe the intended behavior of the system under test, to create an execution trace complete with input and expected output. From these traces, test cases are then automatically generated using a selection criteria. The result is derived by running the tests and comparing with the expected output. An overview of the whole process is illustrated in Figure 4.1. In the following we will explore the two key elements behind the effectiveness of model-based testing, namely the model of the system and the test generation algorithm.

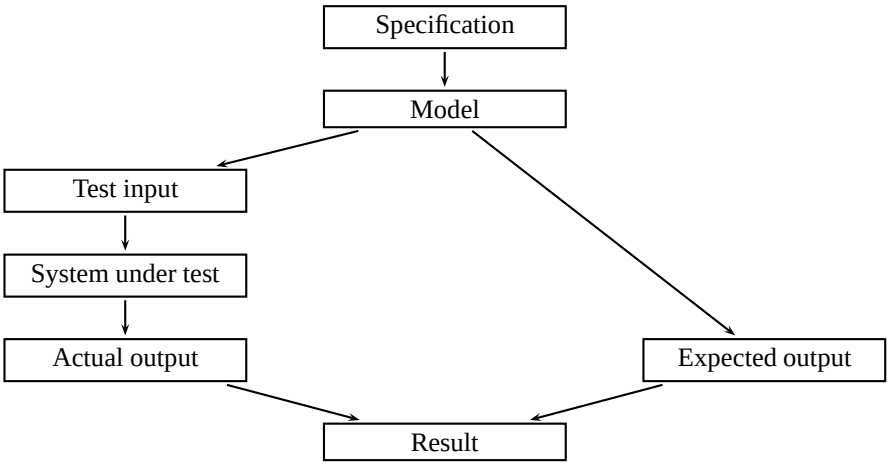

Figure 4.1: The process involved in model-based testing, from how the model is derived from a specification to how the output from the system under test is compared with the expected output from the model.

4.2.1 Models and Abstraction Levels

The model may be extracted from software artifacts or explicitly defined using a modeling language, such as UML, from a specification. However, the modeling language must address different issues than the language used to program the system. For this reason, it may be desirable to use a language, which are more data-driven than a language, such as C.

An important requirement is that the model is formal and rich in detail in order to allow a great amount of automation. A general problem in terms of modeling language is to achieve a good abstraction level, which allows it to focus on system details, such as communication and timing, but also allows it to accommodate for the characteristics of the software artifacts domain or platform. These trade-offs mean that each model is limited in what it can and cannot describe and may lack certain information about the test environment and the system under test.

The level of abstraction has several costs. First, details that are not encoded in the model cannot be tested on the ground of this model[33]. Second, the encoding used for input and expected output derived from the model may be different from the results of real executions. This means that it may be necessary to perform a translation between the two encodings, a process called concretization. Finally, for complex systems the model itself can become quite complex and require model checking to ensure the accuracy of the model.

4.2.2 Generation Algorithm and Selection Criteria

The benefits of model-based testing is the automatic creation of test suites. This makes the choice of test generation algorithm very important. First, different test levels require different test case generation strategies[14]. As an example, model-based unit testing might have to only consider partial models and focus on certain input parameters, while model-based testing of whole systems can focus on operational profiles[13] or more functional specifications. Second, the test generator uses the model to automatically find valid paths through the model. In this sense, test case generation is a search problem, where different approaches can be applied, such as heuristic search or symbolic execution.

A challenge here is to deal with the possible problem of state explosion[10]. This underlines the need of being able to identify the paths that are of most interest and importance. In other words, a very important part of using model-based testing is the selection criteria, which, similarly to test case specifications, help in deriving test cases that exhibit a desired set of characteristics. One possibility is to use concrete criteria, which define specific states of the model that are considered interesting. This can be extended using structural selection, such coverage criteria based on control and data flow analysis of the application, to ensure that many parts of the system is tested. For model-based testing to be effective the generation algorithm also has to consider generating test cases for unintended behaviors. For this, it is necessary to increase the scope of the selection criteria to analyze and identify additional test cases for each transition condition. This can be achieved by employing more random metrics, such as stochastic selection, where probabilistic distribution criteria can ensure that a wide range of different input, which contains both valid and invalid values, is tested. Finally, test adequacy criteria can be used to measure the quality of a test suite as a stopping criteria for the test generation process.

4.2.3 From Models to Test Cases In Practice

To give a concrete example of how model-based testing can be applied, we will limit the scope and focus on presenting a state-based approach. As the specification for the system we will use the state graph in Figure 4.2, which describes a simple application that uses a sense-and-sense strategy to disseminate data. The goal is to be able to generate tests, which at runtime can reveal state transitions faults and corrupt states.

The first task is to describe the system under test in terms of state transition diagrams.

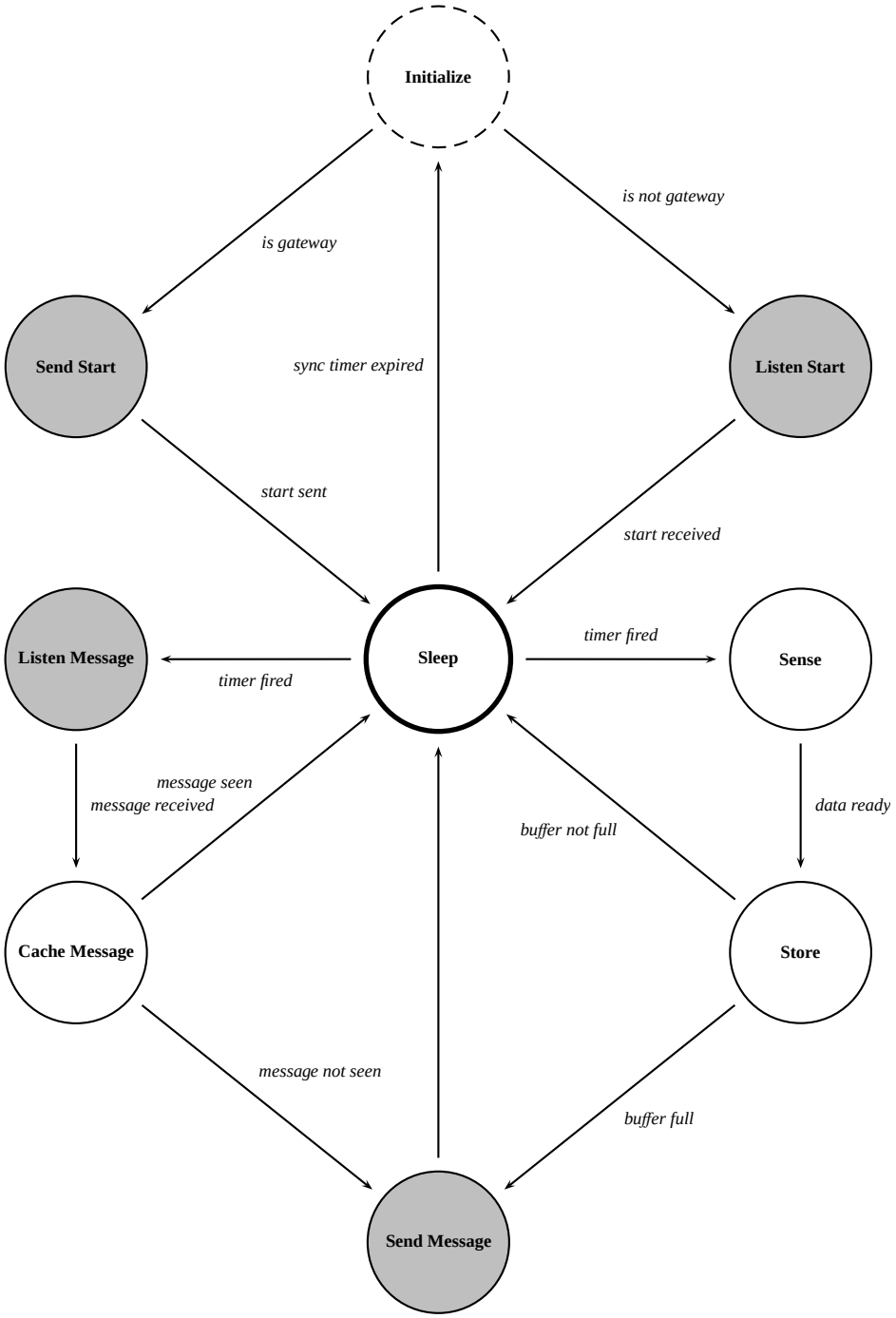

Figure 4.2: The state graph of a single mote for a simple sense-and-send application. The main state is indicated by a bold circle. The color-coded states involve radio communication.

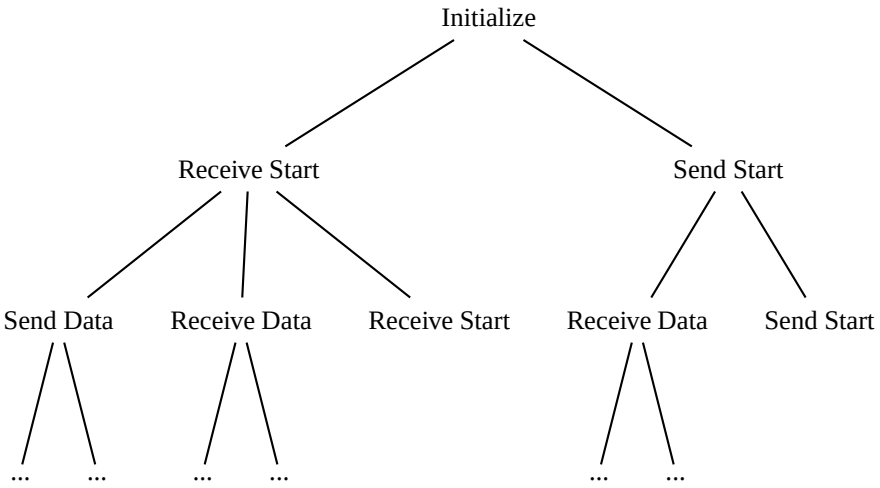

Figure 4.3: Snapshot of a partial transition tree during the test generation phase.

This can be hard due to the concurrency of the application, where routing can be done while sensing is in progress. One simplification here could be to only consider input and output transmitted via the radio. Consequently, each transition defines a state change based on radio communication.

Next, each state transition is annotated with predicates and constraints, which evaluate the context in the model. The predicate are boolean expressions that must evaluate to true for the transition to be valid. Constraint fields allows the number of paths produced to be limited during test generation. During the transformation conditions guarding transitions are evaluated. A third annotation in the form of procedural code may also be added, which can be used to inspect the context during the test execution.

With this model, we are now able to generate the test cases. For the purpose of this example we will use transition tree-based generation with which the state model is transformed into a transition tree as illustrated in Figure 4.3. In the tree, each path from the root to a leaf node is a test requirement for testing the behavior and each leaf node will be a test case template. A naïve algorithm, which does not use any advanced selection criteria, is:

1. The initial tree is the initial state of the state model.
2. Add a node for each valid transition.
3. If the node is a final state or already occurs the node it marked terminal. This eliminates duplicate transitions.

4. Continue until all leaf nodes are marked terminal.

When the algorithm terminates, each test case is derived by concatenating all the test information fields for the path that is transitioned.

4.2.4 Using Model-Based Testing

Model-based testing has been the subject of multiple researches and the findings suggest that the technique can be very efficient, saving up to 80% of the cost of testing[10], and that it is most beneficial when applied to whole-system testing[14]. Furthermore, model-based testing makes test suites less brittle, which is especially effective for systems that change frequently, because tests can be rapidly regenerated after the model has been updated.

The greatest benefits of using model-based techniques lies in terms of automating a very time consuming task and increasing the test coverage. It should be noted that while there is potential for automatically generated tests to cover more possible inputs with fewer test cases[33], the automation provided by model-based testing does not necessarily lead to greater coverage. For example, it is not necessarily the case that the coverage increases simply because significantly more tests can be run. Developing good selection criteria plays an important role in this respect if model-based testing is to perform better than purely randomly generated tests.

While the technique offers a lot of automation, the initial step of creating the model still requires a substantial effort[13]. This can be difficult if the specification makes it hard to determine the constraints of the system. The effort also requires skill and knowledge of the modeling language, development process, and tool chain, which can hinder its acceptance and usability.

For a given system under test both the models of the system and its environment are needed for analysis. Consequently, in the use of models there is also a hidden assumption that the test environment also can somehow be modeled. The extreme dynamics of networked embedded systems might be an obstacle in this respect. It may therefore be more appropriate to evaluate the models at runtime and account for this dynamics. This, in turn, may limit the expressiveness of the models, which can be employed, in order for the evaluation and processing of models in real-time to be scalable.

As a final remark, it remains that there are many open questions that needs further research in order to answer whether or not model-based testing is applicable for testing large networked embedded systems. For example, how much of the process can be automated

and what code artifacts should be considered during test generation. It is also unclear how systems are best modeled in terms of details and expressiveness. Model-based testing, however, has a lot of promising properties, and can be applied in practice, even if various simplification and assumptions are necessary.

4.3 Isolating Software Defects

While the above sections have explored detective measures of using dynamic testing to find failures, the task of diagnosing the failure using corrective measures remains. Based on results from tracing and other methods of instrumenting the system, the error causing the failure needs to be isolated and localized. This can be a very time consuming tasks and it is therefore interesting to explore whether dynamic testing techniques can assist.

4.3.1 Finding Defect Causes

Once we have established that the software has a defect, we can begin to look for its cause. To find a defect, we need to reason backwards, starting with the failure, a task which is very resource intensive[42]. As already mentioned in Section 3.1.4, this task is made harder by the fact that they may not exist a clear link between the defect that cause an infection of the system state and the resulting failure. In the most abstract sense, it is a search in time and space, with time being the execution time where the infection takes place, and space being the state that is affected.

Part of the task of backtracking is to first establish a test case which allows us to reliably reproduce the failure. This can be difficult for non-deterministic programs and long-running programs, where the system state depends on many different events, since it may require control over all possible input sources[42]. With a starting point for triggering the defect, the next step is to further narrow down the test case to exclude unrelated events. In other words, the test case should be simplified so that the defect become clearer and easier to comprehend and communicate. With good programming style, part of the problem of finding the defect becomes more straightforward, since divided and nicely compartmentalized modules are easier to reason about. Finally, we want to be able to automate the test case so that we can reproduce the defect more easily and reliably.

The process of identifying and tracing faults clearly shows that there is a lot of work involved in finding defect causes. We are therefore interested in exploring how test frameworks can help to more reliably reproduce failures and help make the search for a minimal

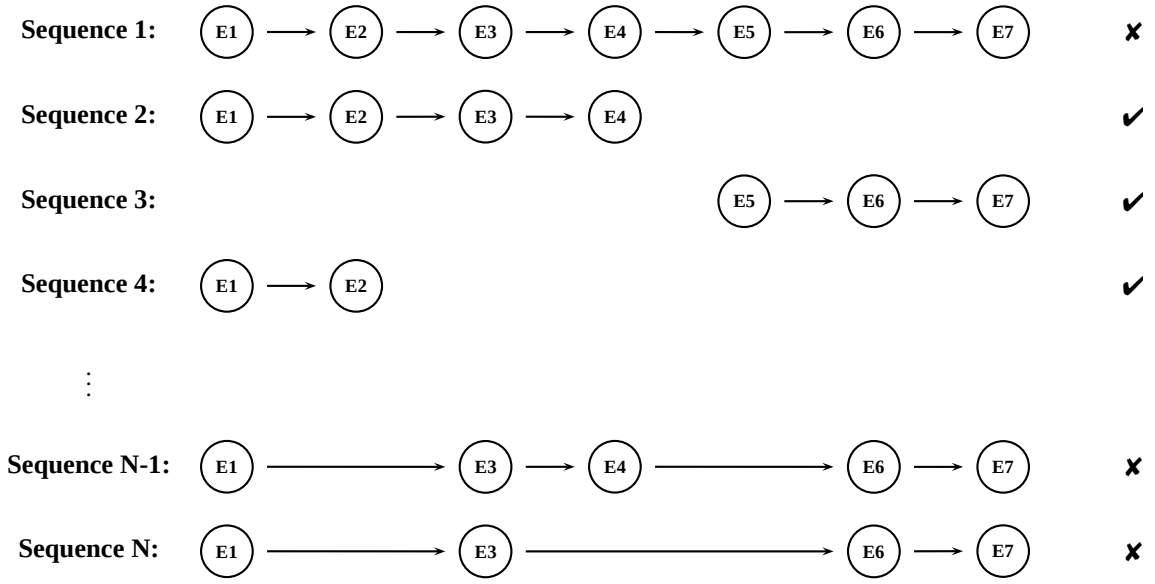

Figure 4.4: Using delta debugging to isolate and simplify failure-inducing event schedule. For each sequence of events the result is indicated by ✔ (pass) or ✘ (failure). Initial a sequence of 7 events (E1 to E7) is known to cause the failure. After N steps the sequence has been simplified to only 4 events.

test case easier.

4.3.2 Isolating Failure-inducing Event Schedules

The main problem in localizing the origin is that in networked embedded system it is necessary to examine the states across multiple motes. For long running applications, the number of states can grow to very large numbers and become unmanageable to reason about. Furthermore, reproducing failures for such long running experiments is not scalable. Consequently, to help identify possible failure causes, a method for simplifying the failure scenarios to the smallest possible sequence of states and events is needed.

One method for achieving this is *automatic delta debugging*[43], an algorithm for systematically producing a minimal set of failure-inducing circumstances. The method can be applied to a broad range of circumstances, such as program input, changes to the program code, or program executions, however, in the following we will consider using it for the isolation of failure-inducing event schedules. In other words, given a sequence of events, isolate the minimal set of events, which are critical for producing the failure.

The basic idea is to systematically narrow down the difference between a passing and

a failing run by gradually simplifying the event schedule. An example of the process is illustrated in Figure 4.4, where initial test runs start by slicing away large subsequences of events and later refine the simplification process to have a finer slicing granularity. At some point, the algorithm will reach a point where the simplification process will no longer yield a smaller set.

As suggested, the efficiency of the whole process is controlled by the narrowing strategy, which is applied in each step. Apart from the approach illustrated in Figure 4.4 of removing events from failing runs, another strategy is to start with the empty sequence and add events until the sequence no longer passes. Finally, the two approaches can be combined so the search for the minimal set will add or remove depending on the result, whereby it can adapt the search strategy to quickly focus on simplifying specific subsequences.

While this rather naïve search approach can be quite effective in finding minimal sets, it has problems similar to the use of random testing. The simplification process may produce circumstances that are inconsistent, and where the test result is neither pass nor failure, but unresolved[41]. One way to deal with unresolved results is to include information about events, which account for properties, such as the relation between events, e.g. event A always follows event B. This information could be extracted from models described in the previous Section 4.2. Apart from reducing the problem of inconsistent sequences, it can also help to make the search for a minimal set more efficient.

4.3.3 Isolating Failure-inducing Code

While finding a minimal test case is a big step forward, the defect still has to be located in the source code. This problem is somewhat similar to the above, in that given the program and the minimal test case we want to isolate which part of the source code is causing the defect. However, where the above approach of slicing and adding works great for simplifying causality chains when it comes to events, it is less applicable for source code. In short, the approach will simply result in considering too many inconsistent programs. A better approach is to use information about the history of changes applied to the software being tested. This information can for example be extracted if the project uses a version control system to keep track of software and the changes they make.

As stated, the delta debugging algorithm can also be used for this, therefore we will look at another but similar approach called *automated bisecting*[12], offered by the git version control system. The tool works by doing a binary search of the history of the source code based on initial knowledge of the last known good version and when the defect

was introduced (usually the current version). For each version being considered, a script is executed, which can compile the source code and run the test case using the resulting program. Based on the exit code of the script, the version being tested is either skipped[1] or marked "good" or "bad", after which the bisecting continues. The process ends when the failure-inducing version has been found.

Because the method relies on using history, the approach is limited to only finding failure-inducing code with respect to regressions, where a previous version worked as expected and later broke. Furthermore, the quality of the result is limited by the size of the changes between each version. In other words, for this method to be effective the code needs to be bisectable meaning each change must be small and well-defined.

4.3.4 Using Software Defect Localization

A prerequisite for using delta debugging and history-based bisection is first of all the ability to reliably reproduce the failure. Furthermore, both methods require a lot of automation of the test process. In terms of resource utilization, they both rely on using a potentially large number of tests, however, the better the knowledge about the structure of the circumstances, be it events or source code, the less tests will be required. To exemplify, if the defect is known to reside in the source code of a specific file only changes involving this file needs to be considered.

While some of the most time consuming parts of localizing defects is automated, the methods still leave manual work for developers. Often, a failure is not the result of a single cause, but rather the outcome of a causality chain of effects and causes[41]. Thus given an initial and simplified event schedule derived from the above method, the programmer still needs to reason about the remainder of the causality chain in order to identify the defect.

4.4 The Case for Dynamic Testing

We believe there is a great potential in using dynamic testing techniques to help developers. The dynamic testing techniques, which has been presented, are very different and can be applied to different parts of the development process. If we accept that different test suites find different errors, it must necessarily follow that a combination of test suites is preferable. This underlines one of the strengths of dynamic testing, because they are able to reuse and extend tests.

[1] E.g. if the test result was inconsistent.

While some of them require a substantial investment of time and development effort, they also promise to provide a richer and more scalable framework for testing networked embedded systems. Finally, several techniques, such as the use of randomized event schedules, can be applied with little effort, which makes them a good starting point for increasing the coverage of test suites.

We conclude this chapter by providing two additional recommendations to those given in Section 3.4.

G: ***The test framework should provide abstractions that allow automated test generation.***

The success of any test suite is its coverage. A good test suite might ensure that no crucial failure will go unnoticed in a very critical part of the code, but if the coverage is incomplete, the infectious behavior of defects can spread from parts with less coverage and threaten the correctness of the critical and well tested components. Because coverage is difficult and time consuming, it is worth investigating ways to automate parts of the test creation process.

H: ***The test framework should support techniques for simplifying test cases in terms of events.***

A natural extension to automated testing is the ability to automatically isolate failure-inducing events in test cases.

Chapter 5

The Re·Schedule Test Framework

To address the issues we have raised in the previous chapters, we present a rudiment design for a test framework called Re·Schedule. It is inspired by concepts from the aspect-oriented programming paradigm and adheres to the guidelines and recommendations presented in Section 3.4. The design is mainly concerned with functionality related with the configuration and control of experiments. While the Re·Schedule Test Framework is designed with the infrastructure of the Re·Mote Testbed Framework in mind, we will not consider specific details related to this association.

5.1 Aspect-Oriented Testing

To help develop a philosophy and guide the underlying approach in the design, we will first introduce the concept of aspect-oriented programming as an inspirational source. Next, we apply the concepts to testing of networked embedded system.

5.1.1 The Aspect-Oriented Programming Paradigm

Aspect-oriented programming[25] is a programming paradigm, which has been developed to address some of the difficulties with clearly expressing certain design decisions using the abstractions and composition mechanisms available in general purpose languages. These design decisions tend to cross-cut the system's basic functionality, whereby they are hard to modularize and end up being scattered through-out the code base and leading to entangled code. Examples of such cross-cutting concerns are logging, security, and error handling.

The primary goal of the aspect-oriented programming paradigm is to provide a mecha-

nism for modularizing these cross-cutting concerns into separate programming units called *aspects*, whereby they become isolated and possible to reuse. The formal definition of when something is an aspect depends on whether or not "it can be cleanly encapsulated in a generalized procedure"[25], be it object, method, etc.

To illustrates the philosophy of the relationship between normal programming units and aspects, we will use an analogy paraphrased from [37]:

> Imagine a mythical world inhabited by dragons and hunchbacks. The hunchbacks all dwell in houses with glass ceilings and work most of their day only communicating by sending messages to each other. Being hunched over, the hunchback are unaware of the existence of dragons, which fly around above them and observe their behavior. The dragons, curious by nature, regularly keep track of the mail correspondence without interfering with it. Occasionally, the dragons leap into action and, for example, repaint one of the houses. The hunchbacks will notice these changes, but continue with their everyday tasks, oblivious to the existence of the dragons.

In the analogy, the dragons represent aspects, which are capable of augmenting the behavior of the rest of the system, represented by the hunchbacks, however, without the need or mean to change the underlying static model of the system. In other words, aspects offer the ability to inject new behavior into an existing system dynamically, transparently and with almost surgical precision[29].

5.1.2 Applying Aspect-Oriented Programming Approaches

In order to apply the concepts given above, we will first reiterate the main challenges we want to address. A distributed application can be viewed as consisting of multiple state-machines, each of which work autonomously. Keeping track of each state is difficult and cumbersome. Furthermore, there clearly exists various parameters of interest to testing, which are cross-cutting the application and not tied to any single mote or components. Finally, as we have pointed out certain aspects of experiments cannot clearly be expressed or is outright cumbersome to express in a general purpose language and can lead to tests becoming brittle.

The basic idea behind applying concepts from aspect-oriented programming to networked embedded systems comes from the observation that the sort of low-intrusion approach of programming by difference, which is proposed in the paradigm, is similar in spirit

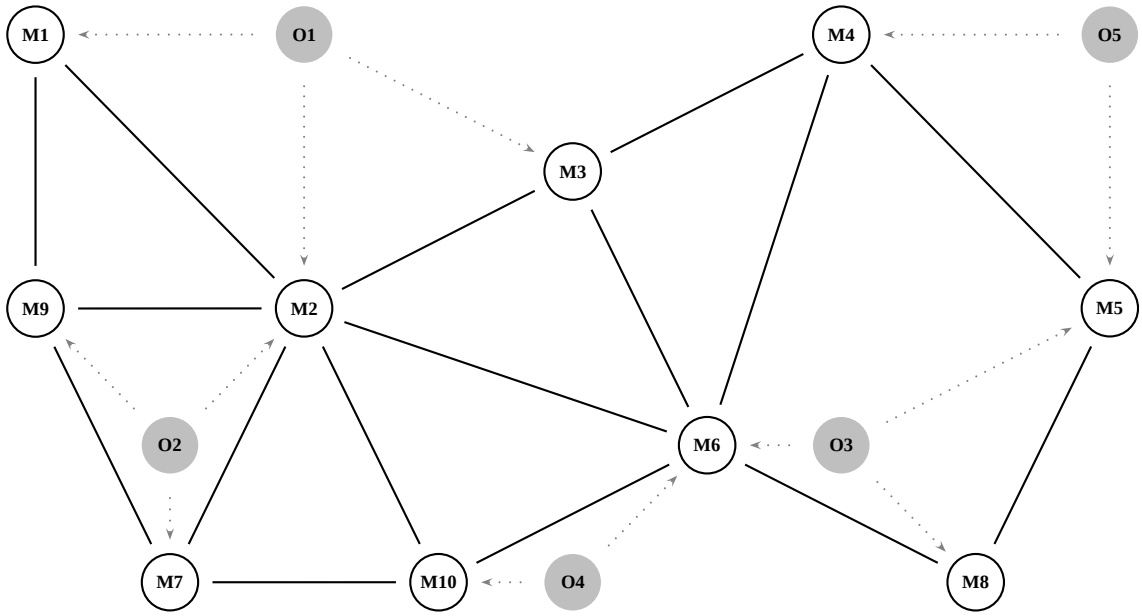

Figure 5.1: Example topology in a testbed with 10 motes (M1 to M10) that takes part of an experiment. Connectivity links between motes are shown as lines. Five additional motes (O1 to O5) represented as color-coded circles observe the experiment by forming an overlay network, which can track and inject messages via the connections represented by dotted lines.

to our desire to observe and modify state for the system as a whole. In this regard the use of aspects offer an additional level of abstraction, where the characteristics of networked embedded system can be build into the model itself.

More concretely, aspects in the form of observable behavior acts as points, where cross-cutting invariants can be inspected by intercepting messages. These observations allow us to trace the application's control and data flow and reason about what is going on. Besides passive aspects that simply track the state, aspects can augment the system behavior by injecting messages or sensor data into an experiment. Consequently, the application of concepts from the aspect-oriented programming paradigm enables a more implicit method of reducing the complexity of testing invariants in the system under test.

To exemplify how the fundamental units of the aspects-oriented programming paradigm can be mapped to networked embedded systems, consider an experiment testing an application for detecting river flooding. This application is concerned with measuring weather conditions such as humidity, temperature and the amount of rain in order to provide input for computing a prognosis. Based on changes in sensor data, the application will change

the frequency of the data acquisition to increase the accuracy. In the example, each of the motes may be considered as components. Similarly, the various tiers and neighborhood of motes can be viewed as components in which information is gathered and shared. Aspects on the other hand is tied to the behaviors of the system, such as the sensor for measuring the rain amount causing the monitoring of the water level in the river to start updating more frequently. The aspects might be defined in terms of a certain message being sent, a certain topology being established, or other observable properties.

An example of the basic setup of experiments in terms of topology and connectivity is illustrated in Figure 5.1. As shown, the majority of motes in the network take actively part in the experiment. Selected motes is assigned the role of "dragons" motes, which together form an overlay network that passively observe and may augment the experiment.

5.1.3 The Case for Aspect-Oriented Testing

We believe that the application of aspect-oriented concept can serve as a framework to for decomposing networked embedded systems into testable behavioural patterns. Furthermore, the ideas naturally allows to account for characteristics of networked embedded systems when testing. As an example, systems capitalize on locality to improve performance. This allows the amount of tracked system state to be reduced by minimizing the experiment to only consider a specific "locality" of the system under test and replaying or simulating the other surrounding parts.

As with aspect-oriented programming, there is however certain properties to have in mind. First of all, aspects while powerful, cannot or should not incorrectly change system state invariants. In other words, no messages can be injected, which violate basic state invariants and assumptions. Second, the concept of aspects presented above is based on the assumption that we can reason about the state of the system as a whole based on behavior. This might not always be the case if the system is treated as a black-box and the system relies on unpredictable behavior. Finally, it relies on the ability to track communication and requires that the application is "networked", since anything that cannot be gathered from tracking the communication, cannot be detected. This is a general problem for event-driven systems and can make it hard to detect certain types of mote failures, for example when duty-cycling is extreme. To conclude, several limitations exist in terms of the type of applications that can be tested using this approach.

While testing inspired by aspect-oriented concepts does not give the level of details that on-board debugging provides, it has a holistic approach as mention in Recommendation A

on page 31. Furthermore, it has the following interesting properties:

Low level of intrusion By definition we are passively observing aspects of the system by tracking the communication. This means that no changes are required for the embedded system in compliance with Recommendation B on page 31.

Reproducing and simplifying experiments is made straightforward In order to reproduce an experiment, we simply use the overlay network to inject messages from a trace of a previous experiment. The task of automatically simplifying a test case, as mentioned in Recommendation H on page 45, is a matter of replaying only a subset of events.

Can scale to large experiments The overlay network only consists of a subset of motes, which makes this approach very scalable, as expressed in Recommendation D on page 31, potentially only restricted by the capabilities of the underlying backchannel.

Abstract system modeling The expressiveness and high abstraction level makes modeling of systems possible. This facilitates automated test generation as mentioned in Recommendation G on page 45.

5.2 Design Overview

The main design goals for the test framework are:

Test of complete systems A main concern is that the framework has a holistic approach, both in theory and practice, which allows testing of complete systems.

Powerful control of experiments To facilitate advanced tests with complex scenarios and timed events, it must be possible to control experiments.

Versatile and extensible Being a framework, it is very important that it is both versatile so it can support a wide range of different work flows and test approaches. Furthermore, to facilitate the exploration of advanced testing techniques it must be extensible.

To accommodate for these goals, the functionality of the test framework is split into 4 major layers illustrated in Figure 5.2. This design captures how the functionality is gradually extended from lowlevel and basic services to increasingly more advanced functionality.

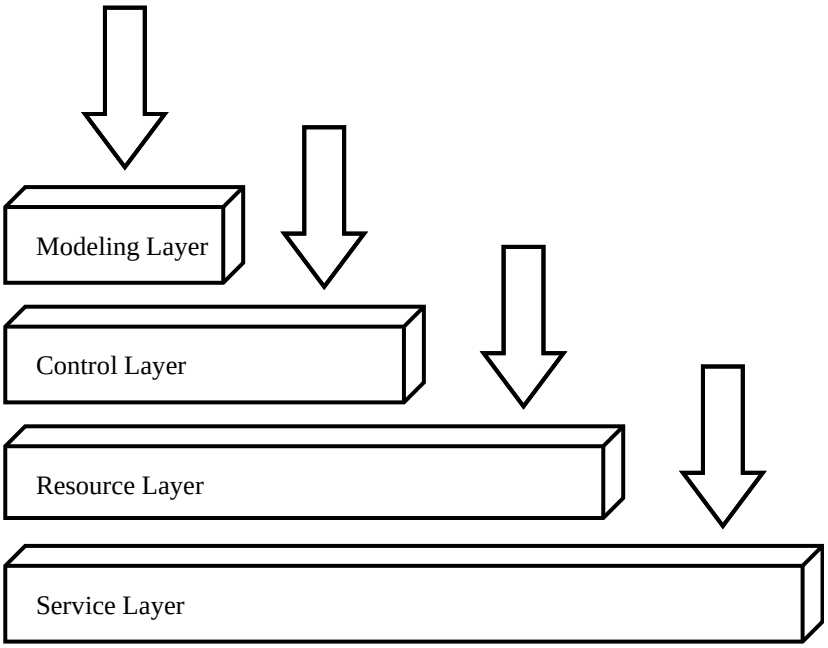

Figure 5.2: The overall design of the test framework in terms of layer.

From an architectural point of view, this allows each layer to confine and encapsulate related functionality and abstractions. Furthermore, the separation enables each layer to be developed independently from each other and provides developers interested in integrating the test framework into applications and tools multiple entry points, each with a different level of abstraction.

The 4 layers are from the bottom and up:

Service layer Which provides very basic access to state information and the different physical entities in the testbed, most importantly the motes. In addition, the service layer can also function as an adapter layer that enables the test framework to run on top of different testbed frameworks.

Resource layer Where physical entities are mapped to abstract resources with the intention of hiding information related to the underlying testbed infrastructure. Considerations for the design of this layer is presented in Section 5.3.

Control layer Which provides the means to control and configure motes and other entities. The design of this layer is presented in Section 5.4.

Modeling layer That extends the underlying layers to facilitate the modeling of systems and their behaviors using high-level abstractions.

5.2.1 Limitations

The design has the following limitations:

Service layer For the purpose of the work presented in the rest of this thesis, we will not address how the service layer should be designed, but simply define it to be the one offered by the Re·Mote Testbed Framework as presented in Section 6.1.1.

Modeling layer While this layer in many ways plays a vital role in enabling the test framework to support holistic and automated testing, a design of the modeling layer is considered outside the scope of this thesis. Our main argument is that defining a design for the modeling layer is a considerable task that must address issues, which are still unclear and requires further research.

Non-interactive Experiments We will mainly focus on the design for supporting interactive experiments, from the perspective that it in itself provides a complete and usable test framework. While allowing users to run experiments locally is valuable and important, a next step is to extend the test framework to handle non-interactive experiments, which are run on a testbed server.

User management policies The test framework is like the Re·Mote Testbed Framework not concerned with how user accounts are managed. Specifically for this area of concern there are many site specific choices to be made as to who has access to what resources provided by the testbed. Furthermore, it is difficult to make assumptions regarding user management because of the level of trust that is involved.

Resource management policies Similarly to user management, it is hard to define a resource management scheme that considers all the different concerns which applies for a testbed. To enforce policies they must be integrated into the service and resource layer. Rather than defining whether specific parameters according to which resource utilization should be optimized, we will simply suggest different alternatives when appropriate.

5.3 The Resource Layer

The main purpose of the resource layer is to decouple experiments and their resource requirements in terms of physical entities in the testbed. The main goal is to make experiments less brittle, because they no longer are tied to infrastructure-specific details of the testbed. Moreover, it might not even be tied to a specific testbed. This sort of generalization is very powerful and seeks to separate the responsibility between users and testbed administrators for the benefit of both. First, the underlying testbed can be changed and upgraded without breaking experiments. Second, if multiple testbeds are available, users can with little or no changes rerun experiments on different testbeds.

Furthermore, the resource layer allows decisions related to resource arbitration to be moved and captured in a central place. By concentrating the decisions, it becomes more straightforward to enforce policies and manage and optimize the resource utilization.

5.3.1 Abstractions

The resource layer introduces two main abstractions: *resources* and *resource claims*. What classifies as a resource depends on the policies and the infrastructure of the testbeds. We will use a general definition, which states that resources are units that can be resolved to either a physical entity or a property related to a set of physical entities. Examples include motes, radio channels, sensors and topology.

The resources used by an experiment are described via a set of resource claims. These resource claims act as constraints, which must be met in order for the experiment to be executed. The process of checking and resolving resource claims into physical entities and properties is called resource mapping. Claims can express hard constraint, such as "all or nothing", or soft constraints that captures more loosely defined mappings, for example "as many of possible" and "5 to 10". Finally, resource claims can also be relative giving the ability to express similar or identical mappings necessary for reproducing an experiments: "same as last time" or using a recorded seed for mapping resources.

To avoid limitations in terms of the type of resources covered by a resource claim, they need to be both extensible and expressive. To achieve this, resource claims are expressed as plain text strings using combinations of key/value pairs. While a small domain specific languages can be used, we will simply use a restricted form of natural language where each line represents a claim, for example:

```
1 radio channel
```

```
5+ motes where platform is dig528-2 and topology is star
```

5.3.2 Requirements

The resource layer depends upon the service layer's ability to provide updated information that enables optimal mapping of resources. This information includes knowledge about the connectivity of each mote, which can be used to form topologies. In turn, the resource layer provides the control and modeling layer with the ability to map resource claims to physical entities in the testbed.

5.3.3 Resource Classification and Mapping

Resource mapping belongs to the class of NP-complete problems[36]. Finding an optimal solution is thus inherently very computationally intensive. It is therefore crucial to apply methods, which can help the search for a 'good enough solution'. Since part of the mapping is related with policies, we provide the design for a general resource mapper.

To optimize the resource mapping a preliminary classification of all resources is first constructed. This way it is possible to quickly reduce the amount of parameters that needs to be considered for each claim. The classification divides resources into groups, each of which has unique features that the actual resource mapping needs to consider. Given a resource classification, satisfiability of resource claims is a matter of checking if each claim belongs to a predefined class. For example a classification can be "mote where platform is dig582-2 and has light sensor".

The use of classifications does not solve the complete problem of mapping resources. For example, a high level constraint such as topology is not easily classified. To address this issue, we reduce topology to the simpler problem of neighborhood: "who is the neighbor for mote X". Using this formulation, topology is straightforward to classify. On top of the neighborhood information, claims related to topology can be mapped.

Since the physical resources in the testbed can change over time, the classification can become invalid. The resource mapper therefore periodically updates its resource classifications. When this happens, all cached preliminary classification results are flushed and recalculated.

5.4 The Control Layer

The main purpose of the control layer is to facilitate advanced experiments with timed events, such as simulated mote failure, injection of sensor data and messages. The goal is to allow configuration and control of experiments, which is able to expressed scenarios interesting for stimulating the system under test. This involves processing of input to the experiment, e.g. in the form of resource claims or sensor reading that must be injected, and processing of output from a running experiment. From the output, the control layer should enable results to be collected to evaluate if the test fails or succeeds and for post-mortem analysis.

5.4.1 Abstractions

The basic abstractions of the control layer is the concept of *jobs* and *job descriptions*. Jobs are a generalization of the different possible types of experiments that is supported by the testbed. We will use a general definition that defines a job as a self-contained unit of work, which is indifferent to whether it is submitted to run interactively or non-interactively. In short, no changes are required for moving a job from an interactive to a non-interactive platform. By this definition, a job can express one or more test cases, a test suite, or even some administrative work, such as testbed health monitoring and topology mapping required by the resource layer.

Jobs are defined via job descriptions, which serves as a formal specification of configuration requirements and job control directives. The main form of configuration requirements is resource claims. Job control directives can be given in the form of code or scripts.

5.4.2 Requirements

The control layer requires that the resource layer maps claims to a list of physical entities encapsulated by the resource abstraction. Alternatively, the control layer can access physical entities directly for extra control. The layer provides functionality to maintain state information related to an experiment and interact with the claimed resources.

5.4.3 Programming Model and Resource Access

To make the control layer portable and allow it to be integrated into application and tools, its basic supported programming model is event-driven and non-blocking. On top of

this minimal model, threaded and blocking models can be created to provide more simple control code.

To allow the control layer to both access resources via the service and resource layer, it uses adapters for communicating with these layers. The adapters are responsible for translating between the abstractions used by the respective layers into the abstractions supported by the control layer.

Chapter 6

Implementation Overview

Based on the analysis and proposed design, a partial implementation of the proposed test framework has been developed. The main goal of the implementation has been to prove the concept of the underlying ideas and to serve as a platform to conduct future work. To achieve this, the implementation focuses on constructing a library that provides the functionality of the control layer and a client in the form of a command line interface, which can be used for running jobs. The implementation uses the existing service abstractions provided by the Re·Mote Testbed Framework as the service layer of the test framework. This way, the implementation can capitalize on the stability of the existing infrastructure, because fewer changes of the server-side is required.

Since the test framework is developed as an extension of the Re·Mote Testbed Framework, this chapter will first present important parts of the Re·Mote Testbed Framework. The rest of the chapter looks at various parts of the implementation with regard to the decisions that have been taken. Finally, some of the known problems and limitations of the implementation are presented to highlight places for improvement and future work.

6.1 The Re·Mote Testbed Framework

While the Re·Mote Testbed Framework was designed with modularity in mind, not all components are easy to extend. Before looking at the implementation details, it is therefore necessary to first get a better overview of the framework in order to figure out which components needs to be changed and address some of the issues that had to be faced.

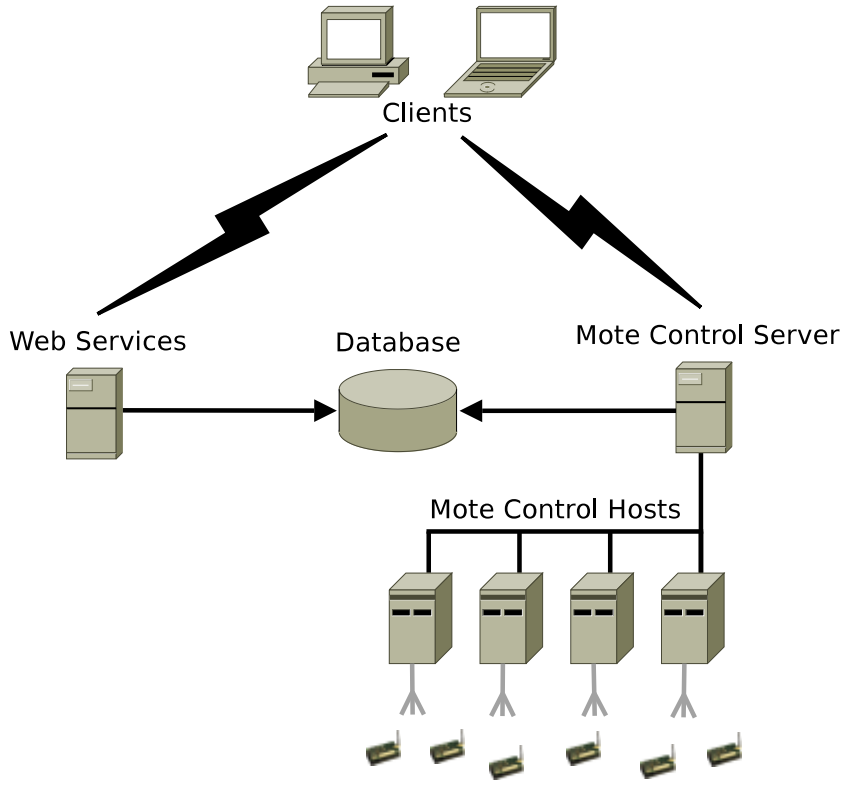

Figure 6.1: The overview of the Re·Mote Testbed Framework. Clients connect to web services and the mote control server. A shared database is used for storing session and configuration information about the testbed. Mote control hosts expose physical mote devices through the mote control server.

6.1.1 Testbed Framework Overview

The Re·Mote Testbed Framework consists of four major components, each with well-defined interfaces and responsibilities. A *database* provides a shared repository for keeping track of the core testbed configuration and session-level state information. This data is used by several *web services* responsible for authorizing users, granting access to motes, and acquiring mote status information as well as the *mote control infrastructure* (MCI), which manages the testbed back-channel. The mote host infrastructure consists of a mote control server to which *clients* connect, and a series of mote control hosts to which motes are connected. The relation between the different components and entities of the framework is illustrated in Figure 6.1.

The framework makes use of many different technologies to accomplish modularity. Java was chosen as the language for the client code, because it very portable and has good

support for graphical user interfaces. Furthermore, many tools and framework exists for Java that makes it attractive, one of which is Axis[1] that is used in the Re·Mote Testbed Framework for providing the web services and generate client code for accessing them.

The mote control infrastructure has been designed to run outside the application server as separate daemons and is implemented in C++. This has enabled this critical component of the testbed framework to better address issues, such as efficiency, security and scalability. The choice of language allows greater access to low-level devices necessary for accessing the physical motes. The architecture of the mote control infrastructure is based on a central server, through which all mote control directives pass. This avoids exposing the physical infrastructure of the back-channel and concentrates the low-level access to a single entity. Custom protocols have been created with C++ and Java libraries for handling communication between clients, mote control server, and mote control hosts.

6.1.2 Extending the Re·Mote Testbed Framework

From the overview given above, it can be seen that there is a clear and natural boundary for the service layer, although it necessarily must arbitrate between web requests and socket operations. In short, the control layer accesses the services provided by a testbed through the web services and the MCI front-end server. To enable this, an adapter must be created, so the underlying access to the web services and the MCI server becomes transparent for the control layer.

The only considerable functional change to the existing testbed framework is related with providing clients a method to infer the order of mote events. This has been done by adding timestamp information in the mote control messages sent by the mote hosts. Accurate time tracking thus relies on the system clock on each of the mote hosts to be in sync. One way to achieve this is by using the Network Time Protocol, however, how it is achieved is outside the scope of the framework.

To enable better integration and reuse of the different components, work has also gone into packaging all components, except the mote control infrastructure, in a Maven[2] repository. Part of this task has been to split the components into modules from which artifacts, such as deployable web applications and extensible client libraries, can be assembled. The motivation behind this has been to enable seamless building of the framework and facilitate future work on the framework. By using Maven, the choice of using Java-based technologies is further consolidated and allows to integrate all the existing Java code from the Re·Mote Testbed Framework.

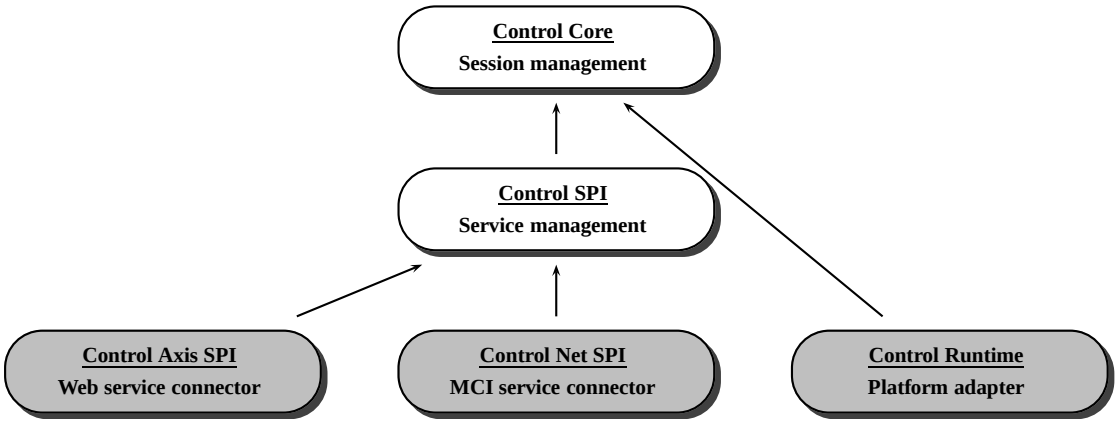

Figure 6.2: Overview of the control library's runtime modules. Color-coded modules are plugable modules that provide platform or application specific functionality.

The rest of this chapter will solely present the implementation of client-specific parts of the test framework.

6.2 Control Layer

The central part of the implementation is the control layer. Its main purpose is to provide transparent access to testbed resources through a uniform interface that enable control of all aspects related to motes. The control layer is implemented as a library.

A general goal behind the implementation of the control layer has been to make it portable across different application frameworks. This means that the resulting library is platform independent. The measure of platform independence has been to allow it to compile as part of an application using Google Web Toolkit[3], a framework for developing rich web applications in Java. For this reason the library depends mostly on self-defined or ubiquitous Java types.

6.2.1 Control Library Overview

The library has been written with modularization in mind and consists of three main modules:

Application programming interface (API) Defines the interface through which clients can interact with a testbed and the motes makes available. For example, commands

are provided for starting, stopping and programming motes as well as reading and writing to the console of a mote.

Core A complete runtime that keeps track of the active session as well as motes in the underlying testbed in order to provide the above API. It also has utilities for maintaining various other state information. To increase reusability of the core module, it uses a platform adaptor for accessing application specific resources, such as work scheduling, and has been kept extensible by providing basic building blocks.

Service provider interface (SPI) Provides an interface for accessing resources through services in an abstract manner. Several service interfaces are provided each of which are modeled after the web services and the commands in the mote control protocol.

To give an idea of how the different modules of the client library are related, Figure 6.2 illustrates an example of what modules are active at runtime. In the example, the core module uses the service manager in the SPI module and an application provided platform adaptor to provide the API. Two SPI modules are used for interacting with the testbed.

6.2.2 Application Integration

Since the implemented control library is envisioned to be part of a larger entity such as a stand-alone application or the control component of a testbed job, it is very important that the it can be seamlessly integrated with the underlying application. It is therefore relevant to go over some of the choices concerning this problem.

The library is not directly concerned with session creation including authentication. The main reason is similar to the reason for using a platform adaptor that applications need to be able to tailor certain parts of the library runtime to make best use of the available resources. For example a web application might have a separate login page, making authentication unnecessary, while other applications might automatically authenticate sessions based on a configuration file.

Furthermore, it depends on a platform adaptor and SPI modules so that the underlying communication and application interaction can easily be changed. This means that it is possible to use one set of SPI modules, which uses sockets and web services, for a client which connects via the Internet, and another set of SPI modules more optimized for a client running locally on the testbed server.

6.3 Command Line Interface

To provide a basic tool for using the library and experimenting with the ideas of the test framework an application in the form of a command line interface has been created. The main goal behind the decision to only offer a command line interface has been to allow the implementation to focus on prototyping the core functionality and be less concerned with usability issues. In the following, an overview of what functionality the tool offers as well as other work related to the application will be presented.

6.3.1 Functionality Overview

The functionality supported by the command line tool is made accessible via several subcommands. They can be divided into interrogative commands and manipulating subcommands. The former provides the ability to query testbed resources, such as listing the available motes. It also allows testbed authentication credentials to be printed and tested, which can be useful for new users when creating a profile for a testbed.

The rest of the commands allows a user to interact with the motes in a testbed and run experiments. For very simple experiments, there is a command for executing a program on a mote, which first programs the mote and then starts it. For more complex experiments there is a command for running job, which can extend the command line interface. The jobs has access to arguments given on the command line and a session, which gives access to motes in the testbed. Several jobs written in JavaScript has been developed, some of which reimplement commands mentioned above. Additionally, there is a command for running jobs packaged as jar file.

6.3.2 Application Libraries

As part of the command line tool, two application-oriented libraries have been created. The goal with this task has been to ease the creation of new tools by offering reusable components. In turn, it has helped to reduce some of the complexity of the created application.

The first library provides core functionality common to all applications. It implements a platform adaptor which can be used together with the client library. Furthermore, it also has support for reading user profiles from configuration files and automatically authenticating sessions. Finally, it contains a factory for creating scripting engines and

setting up the script environment and bindings to core objects common for job experiments, such as the session.

The other library is specifically for handling jobs. In the implementation, jobs are defined as a jar file, since this format is very well supported by the Java standard library. Part of the library is concerned with loading jobs and providing access to the job description and control code. The library also has a simple job manager to interface with the job control code as well as support for control code in the form of scripts.

6.4 Known Issues and Limitations

Although the implementation has been found useful it should be regarded as a prototype. Below, various known problems and limitations are described:

Profiles are mandatory The command line interface requires that the user specifies or creates a default profile properties file. The motivation is that certain testbed properties, such as the URL of the web services, needs to be known upfront to simplify access to the testbed.

Session creation and jobs When executing jobs, the command line interface will always connect to the testbed before execution is started. The reason is that the authentication process is highly application specific and difficult to expose to jobs.

Exiting from jobs Management of the execution life-cycle from jobs is limited. For example, jobs are not able to hook into the runtime of the control layer and receive error notifications.

Printing to standard out Due to how informational messages are handled internally, all errors will be printed to standard out instead of standard error. Furthermore, it can problematic to mix the use of printing from jobs with printing utilities exposed by the command line interface.

Chapter 7

Evaluation and Testing

To evaluate the implementation, this chapter will consider various parts of the system and test their functionality. Each test are founded in the goals and desirable characteristics listed in the introduction, on which a test case based on a small real-world example is described together with expected behavior and results. Jobs are then created for the test cases and the jobs are run on the DIKU Testbed. Problems and unexpected behavior is commented.

7.1 Limitations and Assumptions

Only functional tests are presented in this chapter. Consequently, the user interface itself will not be evaluated. Other parts of the implementation are not tested because they are hard or impossible to test. For example, the parts of the system affected by known problems and issues as listed in the implementation chapter is not considered.

It is assumed that all test cases have access to all motes in the testbed. Furthermore, it is assumed that no external sources interfere with the tests.

7.2 Test Cases

Each test case will give a small introduction of the overall purpose and an explanation of the approach, such as algorithm, and the different artifacts that it uses. Following the presentation is a precise specification of the goal of the test case and expected results. Finally, the results of the test is given along with an evaluation comparing the expected and actual results.

All the test cases presented below have been built using the TinyOS 1.x version for the motes and the applications are based on code developed for a course in networked embedded systems at DIKU. They are ordered so that simpler tests are evaluated first.

7.2.1 Listing Mote Information

The first test case will evaluate the basic functionality of the test framework's client API concerned with mote data. It simply lists all available information about each mote and the host to which it is attached.

Goal and Expected Results

The goal of this test case is to verify that mote information from the database is represented correctly. Specifically, that the relational view sent over the wire via the axis web service is mapped to the more object-oriented view of the client API.

The expected result is that mote data is printed for all motes in the testbed.

Results and Evaluation

The result is as expected.

7.2.2 Programming and Starting a Mote

This test case will evaluate the basic functionality of the test framework API concerned with controlling motes. It programs a mote and starts it. To confirm that the programming was successful, the mote program simply prints something to the mote console.

Goal and Expected Results

The goal is to test that motes can be controlled and more specifically be programmed, and that completion events are signaled properly.

The expected result is to see the output of the mote program.

Results and Evaluation

The result is as expected.

7.2.3 Small Publish-Subscribe Experiment

This test case is based on a small publish-subscribe experiment, where data in the form of imaginary sensor readings is published by a mote and needs to be routed through the network. The same mote program is loaded on all motes, however, to simulate an interesting topology the program contains code for ignoring packets from motes, which are not configured as neighbors.

Goal and Expected Results

The goal is to test that the implementation supports a small real-world experiment resembling what a student on networked embedded systems course might run.

The expected result is to see the motes printing messages to their console to inform about the messages they send and receive.

Results and Evaluation

The result is as expected.

7.2.4 Monitoring Health and Mapping Topology

The final test case is to build a small monitoring job, which can be used to access the overall health of the testbed and help to map the topology of the testbed. The job uses a single mote program, which initially executes some on-mote self tests and then listens for network activity. On request the mote program can be instructed to send a sequence of packages. The general structure of the job is to:

1. Get control of as many motes as possible.

2. Program each motes. To avoid detect problems, set a timeout of 10 seconds after which the mote is marked as defunct and disregarded from the rest of the job.

3. Iterate over each mote and start the mote program. Capture results from the self test via the mote console and log them.

4. Request each mote to send a sequence of 10 packages one per second. Log the result of motes reporting that they received the package.

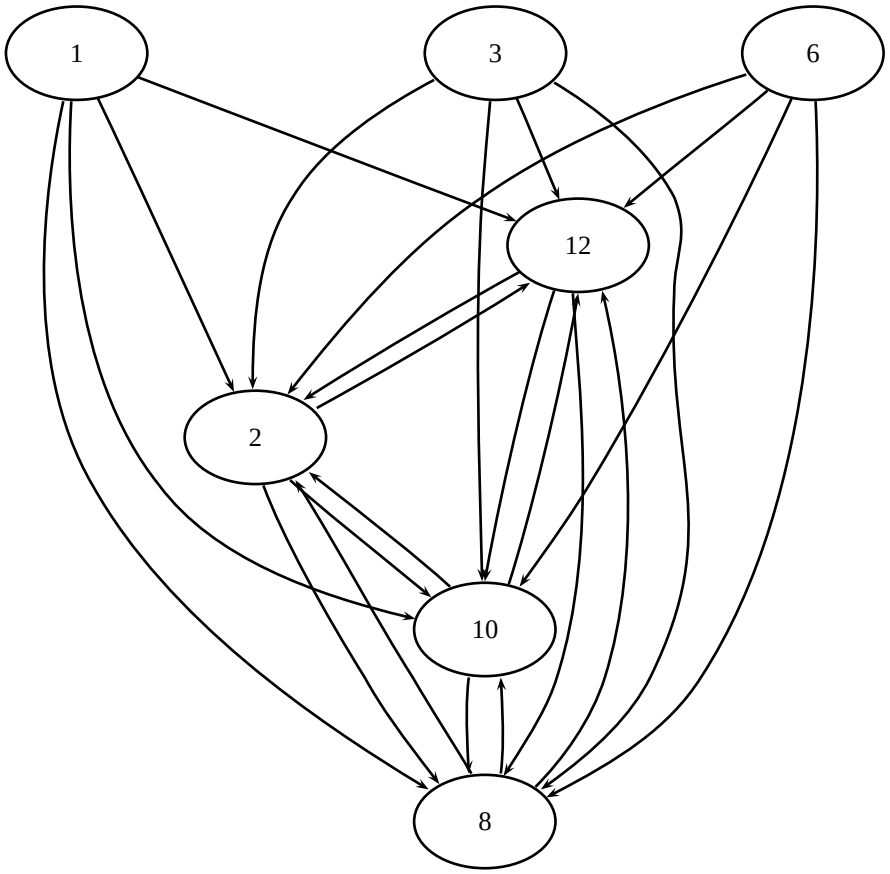

Figure 7.1: The topology mapped by the health monitoring test case.

Goal and Expected Results

The goal is to use all the functionality provided by the implementation to first run a job, second uses multiple motes, third requires advanced control of the job and fourth uses the mote console for on-mote control.

We expect that the application can be used to build a topology map of the testbed with a certain degree of precision. Furthermore, we expect the topology to show that all motes are connected to each other.

Results and Evaluation

The results from running this test case show that it is possible to build a small health monitoring job, which can be used to map the topology of the testbed. Figure 7.1 shows the topology which was mapped during the test. Unexpectedly, all motes are not connected,

which suggests that some of the radios are not capable of transmitting.

There was a few obstacles during the development of the test case, which caused demanded considerable extra effort and increased the complexity of the test script. Most notably, it was necessary to workaround motes, which we were not able to program. This was done using timeouts. Secondly, the desire to execute certain control structures sequentially was complicated by the asynchronous interface.

Finally, in terms of functionality this was the most advanced test case. Considering that the script ended up being around 200 lines of JavaScript code, this clearly shows that the implementated control layer is quite expressive.

7.3 Summary of Results

We have shown that the implemented test framework provides the basic functionality for creating and running both simple and advanced experiments. Most of the scripts are small, which shows that the the interface is expressive. However, the asynchronous program model turned out to make some of the test cases very complex. Furthermore, there are several reoccurring patterns throughout the scripts, which suggests that they should be made part of the API.

The experience from this evaluation shown that a lot of care needs to be taken when developing test cases for the implemented test framework. As mentioned, it was necessary to use timeouts to gracefully handle non-responding motes. This was disruptive and had a very negative effect on the overall impression in terms of the usability of the testbed. It suggests that it is important to be able to continuously and automatically monitor the health of the testbed itself, in order for these issues to not degrade the quality of the service provided by the testbed.

Chapter 8

Conclusion

Networked embedded systems have a tremendous potential for changing the future landscape of computing. One obstacle for reaching this goal, we argue lies in the fact that testing techniques for networked embedded systems are immature and do not accurately capture the needs characterizing these systems. In academia efforts has been made but none has successfully approached the issues on a broader scale. In this thesis, we have addressed the problem by exploring the research space for testing networked embedded systems. In this process, we have analyzed and identified key challenges involved in testing networked embedded system using a problem solving approach. Based on our findings, a test framework has been designed, which we think addresses the fundamental problems and issues. Finally, we have developed a partial implementation of the test framework to evaluate our concepts.

8.1 Discussion

As an initial step to bring a more systematic testing approach to the research field, we have proposed a candidate fault model for networked embedded systems. We think it can serve as a starting point towards more clearly identifying and understanding the origin of faults in networked embedded systems. Moreover, we argue that it must be a key consideration in order to achieve more systematic testing. A challenge in overcoming this task has been to strike a good approach between generalizing without becoming too abstract. We believe this challenge can only be faced by involvement from the research community.

Testing complex networked embedded systems requires detailed knowledge about the state of both the individual systems and their synergy with the surrounding. This has

let us to explore model-based testing approaches, which we argue have the potential to allow test frameworks to more fully embrace the specific challenges of networked embedded systems. We see this as an important step towards more scalable, expressive and effective testing techniques of networked embedded systems. In addition, our findings suggest that more advanced types of dynamic and automatic testing become feasible, because the use of models makes test cases less brittle by reducing dependencies on specific sequences of events and behavior. One thing to keep in mind is that some of the same limitations that exist for simulation also applies for model-based testing, namely that the results are directly related to the accuracy and level of detail of the underlying models.

A lot of possibilities are also available in the use of artifacts. When regarding testing as merely a search problem, the more we know about the various components of the system under test, the more we are able to reduce the search space. From this perspective, artifacts can play a vital role in making testing more efficient. This is especially true for automated testing, where existing methods of randomization and other evolutionary strategies require a lot of resources with only little yield in return. This is a particularly hard and challenging area to approach, because it involve crossing the boundaries of several development processes and practices.

The results from the evaluation shows that part of the test framework is realizable. It has also shown that the robustness depends to a high degree on the underlying testbed infrastructure. Even it its preliminary state, we believe that the implementation can serve as a platform on which to conduct future research of the concepts. We conclude that the design goals has partially been reached with the current platform, but that there is a great deal of work to be done to fully realize the concepts and asses the overall utility of the design and general approach.

We conclude the discussion by following up on the research questions posed in the introduction:

- ***What challenges must be faced when designing a test framework for networked embedded systems?***

  We believe that the analysis given in Chapter 3 provides an good overview of the issues, which must be addressed when creating tools for testing embedded systems.

- ***How do we approach the problem of efficient and systematic testing?***

  With the proposed fault model, we have started the work towards identifying common fault causes. Furthermore, our findings related to the use of artifacts suggests that

they can serve as a valuable basis for evaluating various test criteria, such as coverage and adequacy. Together, we believe that this encourages the development of more efficient and systematic testing.

- ***Can dynamic testing techniques be applied to extends the set of tested features?***

  The analysis of dynamic testing has explored several techniques and clearly shows that they have a lot of potential in dynamically extending tests.

8.2 Lessons Learnt

Testing is a difficult subject to approach because it necessarily needs to impose certain work flows onto developers in order to be systematic and effective. While the task presents itself as straightforward, there are many practical problems and technical challenges that needs to be considered. Furthermore, solutions must deal with several conflicting goals. The lack of interest and success in the TinyOS community for working towards a more unified and general testbed and test framework is a testament to this. Many attempts have been made to create testbeds for networked embedded systems and each has addresses specific parts of the problem area, however, no significant outcome has resulted.

The cause might be that the practical nature of testing makes it a less interesting topic for research. Indeed, many problems are concerned with basic system architecture and infrastructure on a level that requires financial and long-term commitment from institutions and organizations in the research community. From this perspective, to successfully ensure that future research projects are properly accommodated in terms of their requirements for testing infrastructure, we believe a high degree of political involvement is required.

We have also been surprised to find that there is a general lack in the networked embedded system community to look beyond the self-defined boundaries of the research field for inspiration. This may well be founded in its original decision to shed previous research in the quest for redefining computing and embracing the vision of pervasive systems. Our findings suggest that there is a lot of potential in more openly adopting proven and workable approaches from researches in areas, such as aspect-oriented programming.

8.3 Future Work

We identify the following major topics of of future research.

Complete and integrate the test framework with existing tools Work on job and scheduling support must be completed, after which the test framework needs to be integrate with existing tools. Due to the big community surrounding sensor network research, many tools and development frameworks exist. The success of the test framework depends on being present and offered where development is being done.

Extend the fault model The fault model presented in Section 3.3 needs to be extended and adjusted. We propose to make a survey, which investigates different types of systems, both scientific and commercial, in order to identify faults based on practical experience. Furthermore, a thorough study of possible test criteria needs to be done for each of the faults in the model.

Explore dynamic testing techniques Work also remains in exploring how knowledge from different sources, such as source code and runtime artifacts, can be integrated with the test framework. This has potential to provide crucial information, such as test coverage, and facilitate effective dynamic testing. We also believe that model-based testing and how it is best applied to networked embedded systems are important areas of exploration.

Embrace diversity via testbed federations The ability of the test framework to facilitate future requirements of research relies on the underlying testbed infrastructure's ability to embrace both diversity of applications and platforms. To overcome these conflicting goals of versatility and specialization, we think that testbed federation is the only choice. This poses new problems such as resource peering and integration across institutional boundaries, which must be explored.

Moving beyond test frameworks to application frameworks With many of the practical and technical tasks being addressed for providing a solid infrastructure for testing, an interesting topic is to investigate methods in reusing components between test and testbed frameworks and the application frameworks used in deployments. This has the potential to both reduce problems when migrating applications from an initial development infrastructure to the final real world deployment and allow reuse of for example monitoring tools.

Bibliography

[1] *Apache Axis:* http://ws.apache.org/axis/. The Apache Software Foundation, 2000-2005.

[2] *Apache Maven:* http://maven.apache.org/. The Apache Software Foundation, 2002-2009.

[3] *Google Web Toolkit:* http://code.google.com/webtoolkit/. Google, 2009.

[4] Roger T. Alex, James M. Bieman, and Anneliese A. Andrews. Towards the systematic testing of aspect-oriented programs. Technical report, 2004.

[5] Kent Beck. Extreme programming explained: Embrace change, 1999.

[6] Marcus Chang. Evaluation of a web-based sensor network interface. Technical report, 2006.

[7] Robert N. Charette. *Why Software Fails.* IEEE Spectrum: http://www.spectrum.ieee.org/sep05/1685, 2005.

[8] Jatuporn Chinrungrueng, Udomporn Sunantachaikul, and Satien Triamlumlerd. Smart parking: An application of optical wireless sensor network. In *SAINT-W '07: Proceedings of the 2007 International Symposium on Applications and the Internet Workshops*, page 66, Washington, DC, USA, 2007. IEEE Computer Society.

[9] B. N. Chun, P. Buonadonna, A. AuYoung, Chaki Ng, D. C. Parkes, J. Shneidman, A. C. Snoeren, and A. Vahdat. Mirage: a microeconomic resource allocation system for sensornet testbeds. pages 19–28. IEEE Computer Society, 2005.

[10] James M. Clarke. Automated test generation from a behavioral model. In *in Proceedings of Pacific Northwest Software Quality Conference.* IEEE Press, 1998.

[11] Sentilla Corporation. *Metals Manufacturing Energy Savings*. http://www.sentilla.com/pdf/Metals%20Manufacturing%20Energy%20Savings.pdf, 2009.

[12] Christian Couder. *Fully automated bisecting with "git bisect run"*. *http://lwn.net/Articles/317154/*. Linux Weekly News (LWN.net), Eklektix, Inc., 2009.

[13] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 285–294, New York, NY, USA, 1999. ACM.

[14] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: a systematic review. In *WEASELTech '07: Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies*, pages 31–36, New York, NY, USA, 2007. ACM.

[15] 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, New York, NY, USA, 2006. ACM.

[16] D. Estrin, D. Culler, K. Pister, and G. Sukatme. *Connecting the Physical World with Pervasive Networks*. IEEE Pervasive Computing, Vol. 1, No. 1, January-March 2002, pp. 59-69, 2002.

[17] Jonas Fonseca. *Re·Mote Testbed Framework Project: http://remote-testbed.googlecode.com/*. Google Code, 2007-2009.

[18] Jonas Fonseca. *DIKU Testbed: http://testbed.ekstranet.diku.dk/*. Department of Computer Science, University of Copenhagen, 2008-2009.

[19] Jack G. Ganssle. *The Art of Programming Embedded Systems*. Academic Press, Inc., San Diego, California, 1992.

[20] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *In Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323, 2003.

[21] David Gries. *The Science of Programming.* Springer-Verlag„ 1981.

[22] 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.

[23] http://openqa.org/. *Selenium web application testing system.* http://seleniumhq.org/, 2009.

[24] Jason Huggins. *Selenium: The in-browser acceptance testing tool.* http://www.youtube.com/watch?v=78mts_sKNGs, 2006.

[25] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *Lecture Notes in Computer Science 1357*, pages 48–3. Springer Verlag, 1998.

[26] Philip Levis. *T2: What the Second Generation Holds.* http://vimeo.com/3284230, 2009.

[27] Philip Levis and Nelson Lee. *TOSSIM: A Simulator for TinyOS Networks.* http:/www.cs.berkeley.edu/~pal/pubs/nido.pdf, 2003.

[28] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM.

[29] Alan Monnox. *Rapid J2EE(TM) Development: An Adaptive Foundation for Enterprise Applications.* Prentice Hall PTR, 2005.

[30] S.A.A. Naqvi, S. Ali, and M.U. Khan. An evaluation of aspect oriented testing techniques. In *Emerging Technologies, 2005.*, pages 461 – 466. IEEE Symposium, 2005.

[31] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proceedings of HotNets–I*, Princeton, New Jersey, October 2002.

[32] Larry Peterson, Andy Bavier, Marc E. Fiuczynski, and Steve Muir. Experiences building planetlab. In *OSDI '06: Proceedings of the 7th symposium on Operating systems*

design and implementation, pages 351–366, Berkeley, CA, USA, 2006. USENIX Association.

[33] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 392–401, New York, NY, USA, 2005. ACM.

[34] Matt Welsh Deborah Estrin Ramesh Govindan, John Heidemann. Sensor networks in geni. Technical Report GDD-06-19, GENI: Global Environment for Network Innovations, Sep 2006.

[35] John Regehr. Random testing of interrupt-driven software. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 290–298, New York, NY, USA, 2005. ACM.

[36] Robert Ricci, Chris Alfeld, and Jay Lepreau. A solver for the network testbed mapping problem. *SIGCOMM Comput. Commun. Rev.*, 33:65–81, 2003.

[37] Perry Myers T .J. Highley, Michael Lack. *ASPECT ORIENTED PROGRAMMING: A Critical Analysis of a New Programming Paradigm.* http://historical.ncstrl.org/litesite-data/uva_cs/CS-99-29.pdf, 1999.

[38] Jaroslav Tulach. *Practical API Design: Confessions of a Java™ Framework Architect.* Apress, Inc., 2008.

[39] Jaroslav Tulach. *Test Patterns In Java.* http://openide.netbeans.org/tutorial/test-patterns.html, Dec 2008.

[40] Geoffrey Werner-Allen, Patrick Swieskowski, and Matt Welsh. Motelab: a wireless sensor network testbed. page 68, Los Angeles, California, 2005. IEEE Press.

[41] Andreas Zeller. *From Automated Testing to Automated Debugging. http://www.infosun.fim.uni-passau.de/st/papers/computer2000/*. Universität Passau, Lehrstuhl für Softwaresysteme, 2000.

[42] Andreas Zeller. *WHY PROGRAMS FAIL: A Guide to Systematic Debugging.* Morgan Kaufmann, 2008.

[43] Andreas Zeller, Holger Cleve, and Stephan Neuhaus. *Delta Debugging: From automated testing to automated debugging.* http://www.st.cs.uni-saarland.de/dd/, 2009.

[44] Esben Zeuthen. *Re-Mote Testbed Framework.* Department of Computer Science, University of Copenhagen, December 2006.

[45] Pei Zhang, Christopher M. Sadler, Stephen A. Lyon, and Margaret Martonosi. Hardware design experiences in zebranet. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 227–238, New York, NY, USA, 2004. ACM.