Bringing Modern Unit Testing Techniques to Sensornets

Konrad Iwanicki, University of Warsaw
Przemyslaw Horban, University of Warsaw
Piotr Glazar, University of Warsaw
Karol Strzelecki, University of Warsaw

Unit testing, an important facet of software quality assurance, is underappreciated by wireless sensor network (sensornet) developers. This is likely because our tools lag behind the rest of the computing field. As a remedy, we present a new framework that enables modern unit testing techniques in sensornets. While the framework takes a holistic approach to unit testing, its novelty lies mainly in two aspects. First, to boost test development, it introduces *embedded mock modules* that automatically abstract out dependencies of tested code. Second, to automate test assessment, it provides *embedded code coverage tools* that identify untested control flow paths in the code. We demonstrate that in sensornets these features pose unique problems, solving which requires dedicated support from the compiler and operating system. However, the solutions have the potential to offer substantial benefits. In particular, they reduce the unit test development effort by a few factors compared to existing solutions. At the same time, they facilitate obtaining full code coverage, compared to merely 57%–72% that can be achieved with integration tests. They also allow for intercepting and reporting many classes of run-time failures, thereby simplifying the diagnosis of software flaws. Finally, they enable fine-grained management of the quality of sensornet software.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging — *testing tools*

General Terms: Reliability, Measurement, Design

Additional Key Words and Phrases: Unit testing, mock objects, code coverage, software quality assurance, wireless sensor networks, embedded systems

ACM Reference Format:
Konrad Iwanicki, Przemyslaw Horban, Piotr Glazar, and Karol Strzelecki, 2014. Bringing Modern Unit Testing Techniques to Sensornets. *ACM Trans. Sensor Netw.* 11, 2, Article 25 (July 2014), 41.
DOI: http://dx.doi.org/10.1145/10.1145/2629422

1. INTRODUCTION

Unit testing is at the heart of popular software development methodologies, such as eXtreme Programming [Beck 1999], SCRUM [Schwaber and Beedle 2001], and Test-Driven Development [Astels 2003]. What sets it apart from other software quality assurance techniques is the focus on testing each individual module *in isolation* from the rest of the system. The isolation facilitates testing all control flow paths of the module, which greatly improves reliability and is hard to achieve with coarser-grained integration testing [Barrenetxea et al. 2008; Langendoen et al. 2006; Ramanathan et al. 2005; Tolle and Culler 2005]. In other words, unit testing scales well with code. Likewise, due to the isolation, unit testing scales well with the development team, as to write a test, a developer

The presented research was supported mainly by the (Polish) National Science Center within the SONATA program under grant no. DEC-2012/05/D/ST6/03582, and earlier (until June 2013), by the Foundation for Polish Science under grant no. HOMING PLUS/2010-2/4, co-financed from the Regional Development Fund of the European Union within the Innovative Economy Operational Program.
Corresponding author's address: K. Iwanicki, Faculty of Mathematics, Informatics, and Mechanics, University of Warsaw, ul. Banacha 2, 02-097 Warszawa, Poland, e-mail: iwanicki@mimuw.edu.pl.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2014 ACM 1550-4859/2014/07-ART25 $15.00
DOI: http://dx.doi.org/10.1145/10.1145/2629422

need not know the entire system. In addition, unit tests for a module ultimately document the module's behavior and usage, which facilitates collaborative development. Designing for unit testability, in turn, improves code modularity and reuse. There are myriads of other examples illustrating that unit testing is indispensable for producing high-quality software [Astels 2003].

Embedded in physical spaces and operating largely unattended, sensornets do require such reliable software. Yet, numerous failed prototype sensornet deployments evidence that building such software is not trivial [Barrenetxea et al. 2008; Krishnamurthy et al. 2005; Langendoen et al. 2006; Tolle and Culler 2005; Werner-Allen et al. 2006]. In the absence of unit testing, many failures result from the difficulty of pre-exercising corner-case control flow paths in the code to be deployed or from flaws introduced with updates to existing stable code. What is more, recent standardization activities, which aim to make sensor nodes fully-fledged Internet devices, and opportunities for integrating sensornets into complex cyber-physical systems, which these activities offer for industry, will likely make matters worse. Compliance with the emerging standards in combination with practical issues like security, manageability, and interoperability, can yield software surpassing in size any prototypes to date, especially observing the growth in flash memory of low-power MCUs. A massive industrial adoption, in turn, can reshape the software development model: from communal one, in which open-source code is polished by numerous experts, to competitive one, in which proprietary code is built by competing companies. For such companies producing volumes of new code that cannot be disclosed to community experts for review, any means for improving software reliability will be of great value. Unit testing will thus likely become a necessity. Today, however, it is largely underappreciated by sensornet developers: it is far from being a common practice.

It is the main tenet of this article that unit testing is not popular in the sensornet community, because our tools do not sufficiently reduce its costs. Even though the tools, discussed in the next section, automate test execution and management, they lag behind the rest of the computing industry in their support for unit test development and assessment. More specifically, to the best of our knowledge, no tools help a developer *isolate the tested module from the code on which it depends*, which is the very idea of unit testing. In effect, the developer has to resort to integration testing or manually build stub modules that abstract out the dependencies. As we show, the first approach makes checking all corner cases hard while the second is laborious. Likewise, existing tools do not enable *automatically assessing how well tests exercise various control flow paths*. As a result, it is difficult to detect if tests for a corner case are missing, which impairs code reliability. All in all, without appropriate support for developing and assessing tests, unit testing is simply not economic.

In this article, we show how a modern unit testing framework for sensornets can address these issues. As a case study, we use a framework that we have developed for TinyOS 2.x [Levis et al. 2005] and its programming language, nesC [Gay et al. 2003]. While the framework takes a holistic approach to unit testing, the article focuses mostly on the two aforementioned core challenges: (1) minimizing the cost of isolating individual modules and developing tests for such isolated code and (2) automating the identification of control flow paths in the code that are covered by a test. We demonstrate how to address these challenges in the face of resource constraints of sensor nodes. An additional contribution is an empirical evaluation of the presented techniques.

More specifically, to minimize the development costs of tests that exercise a module in isolation, our framework introduces *embedded mock modules*. Mock modules are implementations of interfaces that are automatically synthesized by the framework. They are dynamically configurable, so that they can mimic real modules on which the tested module depends. In effect, they isolate the module from the rest of the system with a minimal effort from test developers. In this respect, embedded mock modules resemble mock objects that form a basis of modern unit testing frameworks for object-oriented languages [Freeman et al. 2004]. However, unlike mock objects, embedded mock modules cannot rely on dynamic language features, such as polymorphism or lazy functions. Due to resource constraints of sensor nodes, such features are not available in the predominant languages for sensornets: C and nesC. In addition, the resource constraints make it hard to provide certain guarantees for mock-based testing from the execution environment. By and large, introducing embedded mock modules poses numerous unique challenges stemming from resource and language

limitations. We demonstrate that to provide elegant solutions, support from the compiler and operating system is necessary.

To enable, in turn, assessing how well various control flow paths are tested, the framework provides *embedded code coverage tools*. The tools automatically instrument code such that it becomes possible to investigate which of its pieces are executed based on various coverage metrics. For example, line coverage allows for each line to determine whether that line is executed during a test. Similarly, branch coverage makes it possible for each condition to infer why the condition is true or false during a test. Again, what differentiates our work from existing coverage solutions are resource constraints and operating system limitations. With existing tools it is virtually impossible to instrument any system more complex than an application periodically blinking LEDs of a sensor node. In contrast, our tools adopt a principle that every bit matters, thereby allowing for instrumenting even large systems. This focus on constraints gives rise to solutions dedicated to sensornets. Like embedded mock modules, we demonstrate how to integrate these solutions into a compiler and how to provide appropriate operating system support.

Finally, having a modern, holistic framework, we empirically confirm the aforementioned qualities of unit testing in sensornets. To the best of our knowledge, these are first such studies. In particular, we show that existing TinyOS tests, which are mostly integration tests, have a low coverage: in the code they target, they exercise only 72% of lines and 57% of branches. Adopting unit testing for individual modules is thus simply a necessity to exercise all control flow paths. However, an average TinyOS module depends on nearly 24 external functions, hence isolating it manually can make the test code surpass the module's code even several times, an overhead that is uneconomic. In contrast, employing mock modules, which automatically abstract out the dependencies, can slash this overhead by a few factors, to a value acceptable by developers, without impairing the code coverage. In general, the solutions introduced by the framework in all aspects of unit testing effectively enable adopting the aforementioned modern software development methodologies in sensornets.

The rest of the article is organized as follows. Section 2 discusses related work. Section 3 gives an overview of our unit testing framework. Sections 4–7 explain in detail the solutions the framework introduces for test development, execution, assessment, and management. Section 8 empirically evaluates the advantages and limitations of these solutions. Finally, Section 9 concludes.

2. RELATED WORK

Software quality assurance in sensornets is a multifaceted topic covering, among others, programming languages [Gay et al. 2003], design patterns [Gay et al. 2007], memory safety [Cooprider et al. 2007], interface contracts [Archer et al. 2007], automated design inference [Kothari et al. 2008], integrated development environments [Burri et al. 2006], and debugging tools [Iwanicki and van Steen 2007; Ramanathan et al. 2005; Romer and Ma 2009; Whitehouse et al. 2006; Yang et al. 2007], to name just a few examples. In this article, however, we focus on testing.

Simulators, such as TOSSIM [Levis et al. 2003], and emulators, such as Avrora [Titzer et al. 2005] or COOJA [Österlind et al. 2006], enable testing complete sensornet systems and protocol suites at scale. Similarly, testbeds, like Motelab [Werner-Allen et al. 2005], Kansei [Ertin et al. 2006], or our own ones [Iwanicki et al. 2008; Michalowski et al. 2012], are invaluable for assessing the performance of complete systems or self-contained subsystems and protocols on actual sensornet hardware. However, while simulation, emulation, and testbed experiments exercise common-case control flow paths well, they do not facilitate testing all corner cases, as the past experience suggests [Barrenetxea et al. 2008; Langendoen et al. 2006; Ramanathan et al. 2005; Tolle and Culler 2005] and this article confirms empirically.

Conversely, formal methods, like symbolic execution [Sasnauskas et al. 2010] and model checking [Li and Regehr 2010], do enable exhaustively analyzing the control flow paths of a system. In effect, they are indispensable for identifying software flaws due to unpredicted interactions between various modules of the system. However, since the system is analyzed as a whole, the state space that formal methods have to explore grows exponentially with software and network size. In effect, to make the verification process feasible, parts of the analyzed code, typically hardware driver code

or complex protocol code, have to be abstracted out [Li and Regehr 2010; Sasnauskas et al. 2010]. Not only is this laborious, but it may also impair the overall software reliability.

All in all, there is no silver bullet in software quality assurance, and thus, unit testing complements the aforementioned techniques. In fact, popular software development methodologies, like eXtreme Programming [Beck 1999], SCRUM [Schwaber and Beedle 2001], and Test-Driven Development [Astels 2003], would be impossible without unit testing.

However, to be economic, unit testing requires holistic support for the entire testing process. More specifically, it requires mechanisms that minimize the effort involved in developing, executing, assessing, and managing tests. When it comes to test development, it is vital to minimize the overhead incurred by dealing with each individual module in isolation from the rest of the system. Ideally, a test developer should focus only on writing minimal test logic for the module. The framework, in turn, should provide an appropriate testing model, should automatically isolate the tested module from other modules, and should combine everything into a meaningful test program that can be executed on a sensor node, in a simulator, etc. Likewise, test execution should be automated to the extent that a single command triggers execution of relevant test cases. Importantly, when a test case fails, the framework should strive to provide descriptive feedback information, thereby sparing the user the effort required to diagnose the failure with a hardware or software debugger. When it comes to test assessment, in turn, the framework should automatically provide information on how well particular tests exercise the various control flow paths and corner cases in each module. Such raw execution information should allow developers to quickly identify any missing test scenarios; aggregated statistics should in turn give managers a measure of quality of their software. Finally, test management requires databases containing test cases and the history of their execution statistics, as well as mechanisms for triggering test execution. Such mechanisms may, for instance, trigger the execution of relevant tests whenever new code is submitted to a repository.

We are aware of two major test frameworks for sensornets: TUnit [Moss 2007] and MUnit [Okola and Whitehouse 2010].

For developers, TUnit defines a simple testing model, in which each test is written as a nesC component with a well defined interface. In addition, it provides a set of assertion macros that the developers can use to verify whether test execution progresses as expected, for example, to check whether a value returned by the tested module is equal to an expected value. A test is executed on a node and communicates with a frontend application running on a PC. The frontend automates the execution process to a large extent, including compiling tests into node images, programming nodes with those images, starting the execution of the tests, and collecting their output. TUnit also gathers simple statistics: the number of successful and failed tests and their duration. Finally, from a managerial perspective, TUnit tests can be organized into suites, and historic results of those tests can be archived and presented in user-friendly reports.

MUnit aims to improve upon TUnit by changing the programming language of test logic: from a language for resource-constrained nodes, nesC, to a high-level language, Python. More specifically, although a module to be tested is still written in nesC, tests for the module are written in Python. The nesC code of the module runs on a node; the Python test code runs on a PC. To combine the two environments, MUnit exports all nesC functions of the tested module into Python prototypes, such that each invocation of a Python function prototype on the PC is transformed into an invocation of the corresponding nesC function on a sensor node through an embedded remote procedure call (ERPC) [Whitehouse et al. 2006]. In other words, a sensor node acts as an ERPC server, whereas the actual testing takes place on the PC and boils down to remotely invoking subsequent functions of the server and checking their results. The transformation of the tested module into a node image implementing the ERPC server is largely automated by the MUnit framework. Apart from that, the overall testing model of MUnit is open.

Taking everything into account, when it comes to test management and execution, TUnit and MUnit offer excellent support. The management functionality runs on PCs, and hence, can be implemented with traditional out-of-the-box solutions, such as databases, code repositories, etc. The solutions for scheduling, running, and coordinating tests between resource constrained sensor nodes

and PCs, in turn, draw from a decade of experience with sensornet testbeds [Ertin et al. 2006; Werner-Allen et al. 2005]. In effect, the functionality offered by TUnit and MUnit for test management and execution is comparable to that of testing frameworks for traditional computer software.

In contrast, when it comes to test development and assessment, TUnit and MUnit lag behind modern unit testing frameworks for computer software. Although they define testing models to ease test development, those models are rather generic, allowing even for system testing. Many aspects are thus left open to test developers, which often implies an additional programming overhead. However, a major development-related drawback of TUnit and MUnit—to some extent recognized by their authors [Okola and Whitehouse 2010]—is a lack of support for isolating code. The frameworks simply require that the tested code is self-contained, which forces developers to deal with any dependencies manually. As we show in Section 8, this incurs a considerable overhead. Likewise, apart from the possibility of manually inserting debug statements, developers have little support from the frameworks when assessing tests. In particular, the frameworks offer no automated means for measuring how well tests exercise the various control flow paths in the code. Overall, the lacking support for test development and assessment likely discourages adopting unit testing at scale in sensornets. Put it simple, before being able to execute tests, one has to develop them; in order to develop subsequent tests, in turn, one should be able to measure if the ones already existing indeed exercise the appropriate pieces of the tested software. The solutions in our framework, notably embedded mock modules and embedded code coverage tools, address these issues.

Modern frameworks for traditional computer software, such as as googlemock [Google 2008] for C++ or mockito [Faber 2008] for Java, handle the code isolation problem with the aforementioned mock objects [Freeman et al. 2004]. However, the dynamic language features necessary to support mock objects are absent in the predominant programming languages for sensornets. Replacing the entire language of a system for the sake of testing may not be an option either, especially considering the resource constraints of sensor nodes. Our embedded mock modules thus demonstrate how such inherently dynamic functionality can be supported on resource-constrained architectures.

The assessment problem for traditional computer software is in turn typically addressed by employing standard code coverage tools, such as gcov [Gough 2004] for GCC or Cobertura [Doliner 2006] for Java. However, as we demonstrate in this article, such tools are based on code instrumentation algorithms that are unsuitable for resource-constrained nodes. Our embedded coverage coverage algorithms thus again illustrate how by employing dedicated solutions and exploiting some trade-offs one can alleviate the resource constraints of sensornets.

3. FRAMEWORK OVERVIEW

Although currently unit testing is not common among sensornet developers, it fits well into a sensornet software development process. There are several ways in which a development team can employ our framework. To illustrate, suppose that the team has to build a (sub)system composed of a number of modules. Assume that before each module is started being developed, interfaces for this module and modules on which it depends are in place. Moreover, there exists a design specifying the semantics of the interactions through those interfaces between the module and its dependencies. In other words, when writing a module, a developer knows what functionality the module has to provide and what functionality it can use, which is a reasonable assumption in our view.

An example of a conservative method of integrating our framework into a development process can be as follows (a corresponding developer's workflow is depicted in Fig. 1). A developer implements a module that provides certain interfaces according to the design. Modules that the module uses to provide its functionality may not exist yet, but their interfaces and behavior are specified by the design, as we assumed previously. When the module (or even just one of its aspects) is ready, the developer can start unit testing. To this end, she first runs a command-line script that generates an empty test suite for the module. Using another script, she generates a template of the first test case, which is automatically added to the suite. At this point, she can start test development by filling in the template with code that exercises a particular usage scenario of the module: typically a particular control flow path within the module's code.

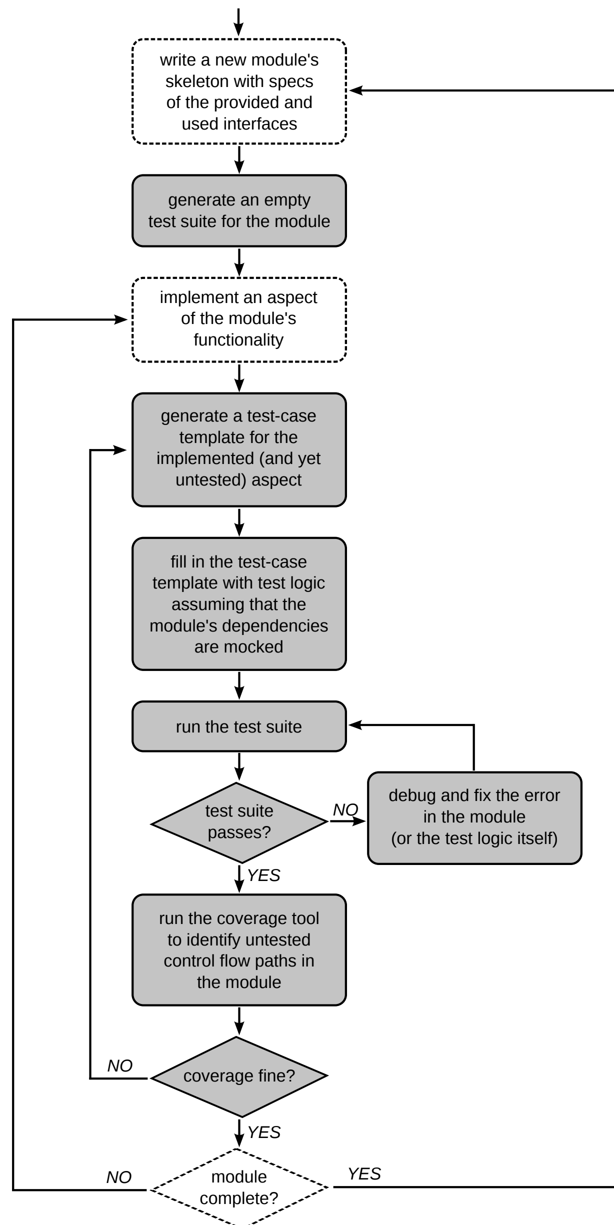

Fig. 1: A conservative development workflow with our framework (a developer's perspective)

The only code the developer has to write for the test case is test logic. The logic boils down to invoking functions of the interfaces provided by the tested module and asserting that these functions behave as specified by the module's design. However, each function of the tested module can invoke functions of the modules on which the tested module depends, and the behavior of the tested function may be determined by the results returned by those invocations. To deal with such dependencies, our framework automatically synthesizes embedded mock modules, which act as replacements of those dependencies, and links them to the test suite. The developer, in turn, just specifies in the

test case which mock functions are expected to be invoked by the tested function and how, and what their behavior should be. In other words, for each test scenario, the developer dynamically configures mock functions to act as their design prescribes in this particular scenario. We elaborate on embedded mock modules in Section 4.

When the test case is ready, the developer deploys and runs it on a node or in a simulator. In fact, she can run the entire test suite with a single command. However, the tested module or the test logic itself may have flaws that manifest as failures during the execution. In particular, the test case may block, some code may incorrectly access memory, a tested function may return an invalid value, fail to invoke the expected mock function, or invoke one that is not expected, to name a few examples. Our framework strives to intercept such failures and provide meaningful feedback to the developer, as we discuss in more detail in Section 5. Presented with such feedback, the developer fixes the flaws and reattempts test execution. This often (albeit not always) saves a lot of time that would otherwise be spent on tracking the flaw with a hardware or software debugger.

After running the test case, the developer should also assess whether it indeed tests the selected control flow path of her module. To this end, she executes our command-line coverage tool for the module, which produces a report with the selected coverage metric, as we explain in Section 6. The line coverage metric identifies all source code lines of the module that were executed in the test, thereby allowing the developer to infer the flow of control in the module during the test. The branch coverage metric, in turn, becomes useful when a given control flow can be triggered by several conditions (e.g., an alternative in a conditional statement), and the developer wants to confirm that the flow was triggered by a particular condition. If the coverage report confirms that the test case exercises the selected control flow path, the developer proceeds to write another test case; otherwise, she revises the current one. Moreover, the developer can use the coverage report to verify that she is not missing a test for any corner case in the module: the source code lines or conditions corresponding to such a corner case would simply be marked as uncovered in the report.

Finally, the framework is also used to facilitate test management, which is especially important considering that a code repository is dynamic: new code is introduced, existing code is modified and removed, and even module designs change. Since embedded mock modules reduce the testing code that must be written to pure test logic, the team may be required to produce unit tests for every developed module. Upon committing any code to the repository, relevant test cases are triggered automatically, and, if any of them fails, the changes are rejected with appropriate feedback. Similarly, the embedded coverage tools may be used to verify that no committed code has fragments uncovered by unit tests. The overall effect of such an approach is that the resulting software, even in the face of constant changes, is less likely to have undetected defects, which is important for deeply embedded sensornet systems. We elaborate on test management with our framework in Section 7.

Another sample approach to employing our framework in sensornets, which is far more radical than the one discussed hitherto, is test-driven development (TDD) [Astels 2003]. TDD is an agile methodology that turns the conservative software development process around. It relies on the following short cycle, potentially involving more than one developer (cf. Fig. 2). Rather than writing the code of her module, a developer first writes an (initially failing) test case that defines a single desired aspect of the module's interface. Only then does she (or another developer) produce a minimal amount of code for the module to pass this test case. Finally, the code is polished to meet the team's standards and the process is repeated (possibly with a different developer configuration) until all aspects of the module have been implemented. Intuitively, TDD may be viewed as a game between test and module developers in which the test developers are always the first ones to move.

TDD is heralded for encouraging the simplest possible implementations and reinforcing developers' confidence in their code. It has thus recently been gaining popularity in businesses where software reliability is at stake. However, it is neither effective nor economic unless supported by a holistic unit testing framework, such as ours. In particular, embedded mock modules ensure that writing a test case requires a minimal effort, even when a large fraction of modules on which the currently developed module depends have not yet been implemented. Highly automated test execution and descriptive failure feedback minimize in turn the time required to discover why a test

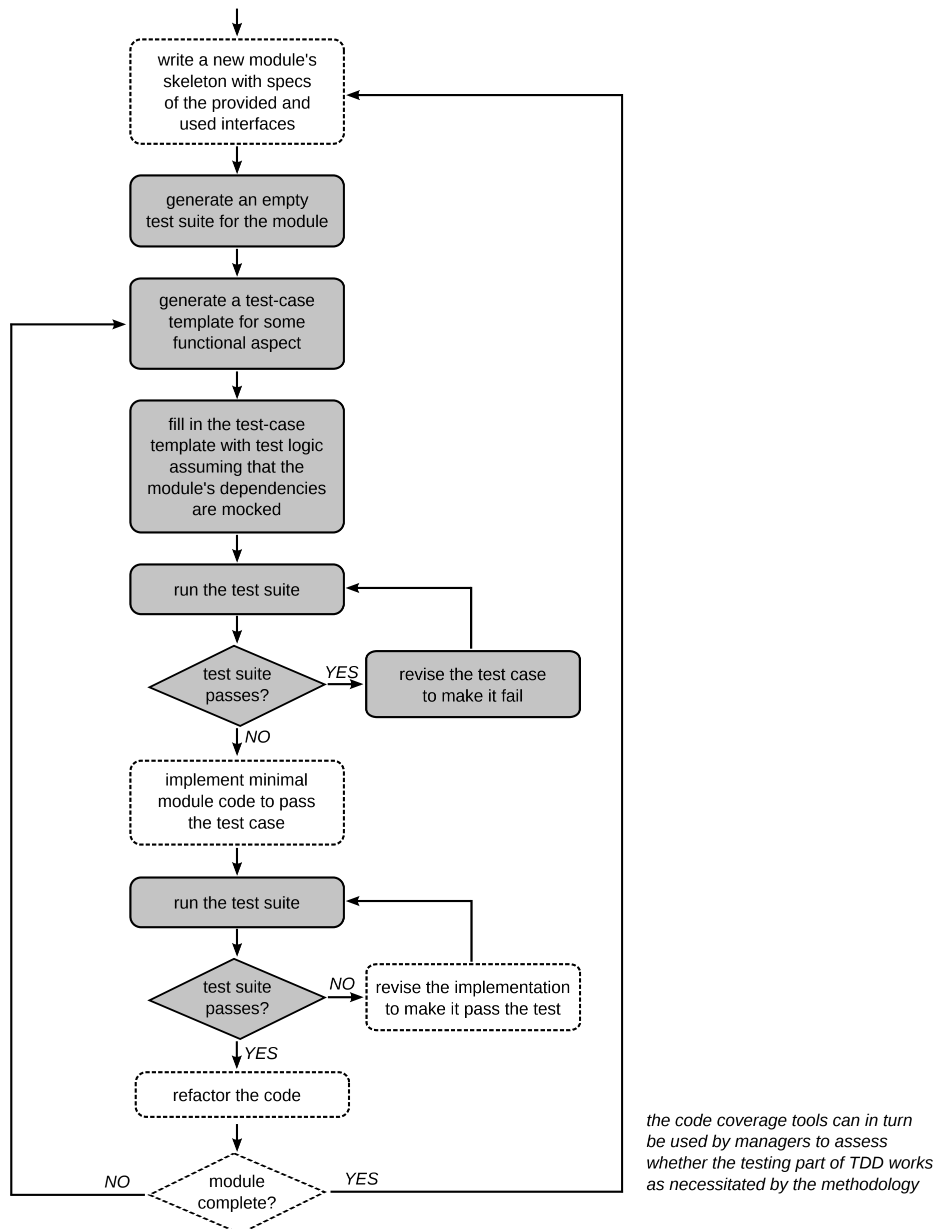

Fig. 2: A test-driven development workflow with our framework (a developer's perspective)

case does not pass. Finally, embedded code coverage tools allow for more informed decisions when devising a next test case that should fail on an incomplete implementation of the module.

All in all, we believe that, rather than forcing a particular one, our framework can enable a variety of software development methodologies in sensornets. It brings to sensornets a set of modern unit testing mechanisms, which can be used in a development process tailored to individual users. We elaborate on these mechanisms and the challenges they pose in sensornets in subsequent sections.

4. DEVELOPMENT PERSPECTIVE

In our framework, unit tests are developed in the same language as normal code. We have noticed that a single programming language is convenient, particularly for methodologies such as TDD, which involve frequent switching between writing module and test code. Since the framework is meant for TinyOS 2.x [Levis et al. 2005], this programming language is nesC [Gay et al. 2003]. Nevertheless, most of the presented problems are fundamental: they are inherent in other sensornet development environments [Akhmetshina et al. 2003; Dunkels et al. 2004]. Likewise, while our solutions are illustrated with examples in nesC, adapting them to other environments should be fairly straightforward, as we explain throughout the article.

4.1. Basic Development Support

Like TUnit, MUnit, and frameworks for traditional computer software, as a starting point for developers, our framework specifies a test case interface. The interface defines an entry point to test code, a `startTest` function, which acts much like the standard `main` function in C programs. Having a custom entry point allows for combining multiple test cases into a single test application, a *test suite*, thereby making test execution more efficient. Furthermore, before invoking the custom entry point of a test case, the framework automatically initializes various commonly used components, including node communication subsystems, error reporting libraries, and mock modules. Such an approach spares developers the effort involved in manually configuring the execution environment for each test case. Finally, the test case interface combined with a well defined execution model allows for generating test case templates automatically by a command-line script. In effect, the developer has to just fill in the templates with pure test logic.

When it comes to the test logic itself, it essentially boils down to invoking functions of the interfaces provided by the tested module, usually in a specific order, and asserting that these functions behave as specified by the module's design. One aspect of the tested behavior is the result returned by a function. Another is how the function changes those of its parameters that are passed by reference. Finally, a change in the module's internal state caused by the function is equally important.

Again like in TUnit, MUnit, and frameworks for traditional computer software, in our framework, testing these aspects is facilitated by a family of assertion macros. Such a macro checks a given condition at run time and, if the condition is not satisfied, terminates the current test case and reports an appropriate error. Listing 1 presents sample uses of assertion macros. It shows that they allow for succinctly expressing the tested property. However, as we discuss in Section 5, while conceptually simple, assertion macros pose several implementation problems stemming from the combination of resource constraints of sensor nodes and the run-time guarantees our framework provides.

```
1  // Checking return value.
2  TM_CHECK_EQ(call Radio.start(), SUCCESS);
3
4  // Checking parameter modifications.
5  call Mutex.init(&m_mutex);
6  TM_CHECK(! m_mutex.lock);
7  TM_CHECK_NULL(m_mutex.thread_queue.l.head);
8
9  // Checking state modifications.
10 call Timer.startOneShot(16 * 1024);
11 TM_CHECK(call Timer.isRunning());
```

Listing 1: Sample assertion macros facilitating testing function behavior

4.2. Embedded Mock Modules: Motivation

Even though the assertion macros and automatically generated test case templates help programmers to some extent, they do not address the core problem of unit test development: isolating the tested

module from the modules on which it depends. A quantitative study confirming the gravity of this problem is given in Section 8. Here, in contrast, we focus on its nature and our solutions.

Consider a module that periodically reads a sensor, puts the reading into a packet, and transmits the packet over the radio. If the transmission is successful (i.e., `SUCCESS` is returned by a send operation), the entire process is repeated in the next iteration of an endless loop. However, if the transmission times out, (i.e., a particular error code, `ETIMEOUT`, is returned by the send operation), the module backs off for a random time interval and reattempts the transmission.

The send operation the module uses is implemented by lower-level modules that constitute the radio subsystem. Consequently, the module depends on those modules. In particular, whenever it is to be tested, these dependencies have to be dealt with. Without our framework, essentially two options are available: testing the module together with the dependencies or implementing so-called stub modules instead of the dependencies. Both these options have major drawbacks.

Testing the module with dependencies—integration testing—is difficult if all control flow paths in the module are to be exercised. To visualize, let us refer to the left side of Fig. 3. Assume that the dark gray box corresponds to the tested module. The remaining solid boxes represent in turn the radio subsystem. Exercising the control flow path with a successful transmission is arguably easy if the radio subsystem works as expected most of the time. In contrast, testing the timeout path is far more involved. First, the test developer has to analyze the entire radio subsytem to check where the send operation branches to return `ETIMEOUT`. Such an analysis may be laborious, because it involves many modules and context switches on interrupts generated by radio hardware. Second, the developer has to invent how to trigger the branch in the control flow. This may require preparing a packet in a special way, which may be tricky, as the packet crosses many modules that can modify it on the way. Worse yet, it may be necessary to bring the various modules into a certain state, which may be arbitrarily difficult, as the modules interact with each other and hardware. By and large, integration testing is costly and rarely covers all control flow paths. In addition, it can be used only for modules for which dependencies have already been implemented, which impairs collaborative development and is at odds with methodologies such as TDD.

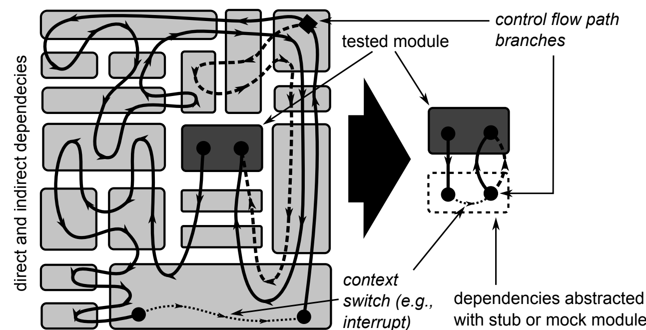

Fig. 3: Abstracting out dependency helps manage complexity when testing all control flow paths

In contrast, abstracting out dependencies eases testing all control flow paths considerably, as visualized by the right side of Fig. 3. The entire logic of dependencies can be reduced just to the paths relevant to the tested module, and the paths themselves can be the simplest possible. In our example, the abstraction of the radio subsystem could be as simple as a returning either `SUCCESS` or `ETIMEOUT` as the result of the send operation, perhaps from a different context, depending on the underlying operating system. However, as we show in Section 8, manually implementing the abstracted interfaces by means of stub modules is tedious and dominates the cost of unit test development, effectively making unit testing uneconomic.

4.3. Embedded Mock Modules: Principal Idea

Embedded mock modules, introduced here, abstract out dependencies but, at the same time, address the issues of stub modules. However, before presenting them, let us slightly formalize the concepts of interfaces and modules, which we have used intuitively hitherto.

In TinyOS, an interface is a nesC file that contains a list of function declarations describing given functionality and preceded by keyword `interface`. However, the concept of interface is inherent in virtually all other sensornet development environments. In particular, in C-based environments [Akhmetshina et al. 2003; Dunkels et al. 2004], interfaces correspond to `.h` files with function declarations.

In contrast, a module is a nesC file that starts with keyword `module` and contains actual code implementing one or more interfaces. Each module explicitly declares the interfaces it provides and uses with keywords `provides` and `uses`, respectively. It can also declare private state and functions. Again, the concept of module is prevalent. In particular, in C-based sensornet development environments [Akhmetshina et al. 2003; Dunkels et al. 2004], a module is a single `.c` file. It provides those interfaces whose function implementations its contains. By symmetry, it uses those interfaces whose header files it includes. Its private state and functions are those variables and functions that are not declared as external.

Although throughout the article we use the word "module," what the framework actually allows for testing is components. In nesC, a component is either a module or a so-called configuration. A configuration is a file that starts with keyword `configuration` and contains no code, but just wires a collection of modules into a larger abstraction, such that an interface provided by one module is wired to an interface used by another module in the configuration. The configuration can also provide and use an interface externally by exporting a corresponding interface of one of its modules. In other words, whether a component is a module or a configuration is largely transparent in nesC. A similar effect of composing modules can be achieved in other environments. In particular, in C-based ones, rather than a single `.c` file, multiple ones can be considered. The used functions would be those ones that appear in the header files, but are not defined in any of the `.c` files. The goal of such module composition is allowing for arbitrarily selecting the unit of testing: small units corresponding to individual modules make it easier to test all control flow paths; in contrast, larger ones enable testing interactions between modules. In the remainder of the framework description, the word "module" thus in fact denotes a component.

In any case, rather than a particular convention, what is important for our ideas, and unit testing in general, is that a system has to be decomposed into modules (components), which need not be self-contained but for which the only means of interaction are interfaces. As a result, modules can be substituted at compile time without affecting other modules. Since this is a fundamental idea behind most programming languages, while the solutions we present here are implemented for TinyOS, the ideas behind them are broadly applicable.

The operation of embedded mock modules can be summarized as follows:

(1) The developer fills in test case templates for a module with logic exercising the interfaces the module provides. She assumes that the interfaces the module uses in its implementation have been mocked, that is, they are implemented by embedded mock modules.
(2) Under this assumption, in each test case, the developer must be able to adapt the behavior of the mock modules to the control flow path exercised by this test case. To this end, her test logic sets *expectations* regarding the use of the mocked interfaces by the tested module and *actions* the mock modules will perform in response to the invocations of their interfaces.
(3) The framework automatically identifies the interfaces used by the tested module and generates implementations of mock modules that provide these interfaces.
(4) The framework also automatically generates additional (hidden) configuration and verification interfaces of the mock modules, a subset of which the developer used in her test logic.
(5) The framework links everything, including the test case logic, the aforementioned generated code, and other components into a test application that can be run on a node or in a simulator.

(6) At run time, the test case logic invokes an interface of the tested module to verify its behavior. The tested module may in turn invoke the interfaces provided by the mock modules.
(7) Upon an invocation of its interface, a mock module verifies the expectations for this interface. If an expectation matches the invocation, the corresponding action is performed; otherwise, an error is reported.
(8) At the end of the test case, the framework automatically verifies whether all the configured expectations have been satisfied for all the mock modules.

Let us illustrate these principles with examples. Listing 2 presents a simplified start sequence for the aforementioned module (Section 4.2). The code initializes a random number generator (line 2), which is used, among others, for generating back-off delays, starts the radio (line 5), calibrates the sensor with two coefficients (line 8), and starts a timer (line 11) that determines when the sensor is read. Each of these operations may fail, in which case the entire start sequence should fail.

```
1  command error_t StdControl.start() {
2    if (call Random.init() != SUCCESS) {
3      return FAIL;
4    }
5    if (call Radio.start() != SUCCESS) {
6      return FAIL;
7    }
8    if (call Sensor.calibrate(COEFF_A, COEFF_B) != SUCCESS) {
9      return FAIL;
10   }
11   call Timer.startOneShot(1024UL * 60UL * 5UL);
12   return SUCCESS;
13 }
```

Listing 2: Initialization sequence of sample sense-and-send module from Section 4.2

Listing 3 presents the entire test code that has to be written to exercise the control flow path in which an initialization failure of the random number generator causes a failure of the whole start sequence. The new constructs for embedded mock modules are underlined.

```
1  expect command Random.init():WillOnce(Return(EINVAL));
2  TM_CHECK_NE(call StdControl.start(), SUCCESS);
```

Listing 3: Entire test code for first failure path in code from Listing 2 involves one expectation

In line 2, the code invokes the sequence and checks with an appropriate assertion macro that the sequence indeed fails. Before, however, it instructs the mocked random number generator that its `init` function is expected to be invoked once and, as an action, it should return an error, *EINVAL*. When the start sequence invokes the `init` function of the random number generator (line 2 of Listing 2), the mock module intercepts the invocation and returns the error. In effect, the initialization sequence will indeed fail (line 3 of Listing 2), meaning that the test case will pass. If, in contrast, the developer forgot to put the initialization code for the random number generator in the start sequence, the expectation would be unsatisfied; each such expectation automatically causes a test case failure.

Listing 4 shows the test code for the second failure path. This time the random number generator initializes correctly, but a failure starting the radio should terminate the sequence. The scenario requires changing one expectation and adding another one.

Expectations can be more complex. To begin with, the invoked function may take arguments, and a test case may require checking the values of these arguments. For example, Listing 5 presents the test code for the third failure scenario, in which sensor calibration fails.

```
1  expect command Random.init():WillOnce(Return(SUCCESS));
2  expect command Radio.start():WillOnce(Return(EBUSY));
3  TM_CHECK_NE(call StdControl.start(), SUCCESS);
```

Listing 4: Entire test code for second failure path in code from Listing 2 involves two expectations

```
1  expect command Random.init():WillOnce(Return(SUCCESS));
2  expect command Radio.start():WillOnce(Return(SUCCESS));
3  expect command Sensor.calibrate(UEq8(COEFF_A), UEq8(COEFF_B))
4      :WillOnce(Return(EINVAL));
5  TM_CHECK_NE(call StdControl.start(), SUCCESS);
```

Listing 5: Use of argument matchers to test third failure path in code from Listing 2

The expectation for the sensor mock module (line 3) contains *argument matchers*, `UEq8`, for the two sensor calibration coefficients. An argument matcher is a function that takes the actual value of an argument and returns a boolean. In particular, `UEq8` takes an unsigned 8-bit integer and compares it with the value given in the expectation. If the values are equal, it returns true; otherwise, false. As a result, if the initialization sequence calibrated the sensor with coefficients different than `COEFF_A` and `COEFF_B`, no matching expectation would be found, and the test case would fail.

Apart from having correct arguments, it may also be vital that functions be invoked in a specific order. Suppose that in our example the initialization of the random number generator must be done first, the timer must be started last, and the radio and sensor must be started in between, but their relative initialization order is irrelevant. To this end, expectations can be organized into sequences, as shown in Listing 6. In the listing, if any of the expectations is triggered out of order in any of the sequences to which it belongs, the entire test case fails.

```
1  Sequence * s1 = NewSequence();
2  Sequence * s2 = NewSequence();
3  expect command Random.init()
4      :AddToSequence(s1):AddToSequence(s2)
5      :WillOnce(Return(SUCCESS));
6  expect command Radio.start()
7      :AddToSequence(s1)
8      :WillOnce(Return(SUCCESS));
9  expect command Sensor.calibrate(UEq8(COEFF_A), UEq8(COEFF_B))
10     :AddToSequence(s2)
11     :WillOnce(Return(SUCCESS));
12 expect command Timer.startOneShot(Any())
13     :AddToSequence(s1):AddToSequence(s2);
14 TM_CHECK_EQ(call StdControl.start(), SUCCESS);
```

Listing 6: Use of sequences to ensure ordering on success path in code from Listing 2

If just one expectation is concerned, the number and ordering of invocations can also be controlled with various *modifiers*. Listing 7 presents examples of selected modifiers. Example (a) illustrates granting exclusive access to a resource. The reservation function is expected to be invoked at least twice. The first reservation will be successful. The second one will inform that the resource is in use. Any subsequent ones will not be treated as failures, but will also politely inform that the resource is in use. Example (b) shows an output FIFO with a limited capacity. The expectation for the writing function allows for at most 64 invocations, meaning that no more than 64 bytes can be written to the FIFO. Any subsequent write will fail to match the expectation, thereby causing an error. Example (c) involves two non overlapping expectations for the same function. The first one can be matched an arbitrary amount of times, possibly zero; the second, at least once. Finally, example (d) shows overlapping expectations for a timer setting function. The first invocation of the function will be

correct and will match the second expectation. Any subsequent ones, will cause errors, because they will always successfully match the second expectation, but the (default) action for that expectation has already been triggered by the first invocation, and by default an action is triggered only once. This differs from example (a) in which there is a single expectation, but with multiple actions.

```
1  // Example (a)
2  expect command Resource.reserve()
3      :WillOnce(Return(SUCCESS))
4      :WillOnce(Return(EBUSY))
5      :WillRepeatedly(Return(EBUSY));
6
7  // Example (b)
8  expect command FIFO.writeByte(Any())
9      :AtMost(64);
10
11 // Example (c)
12 expect command Packet.getPayload(NonNull(), UGt8(108))
13     :AnyNumber()
14     :WillRepeatedly(Return(NULL));
15 expect command Packet.getPayload(NonNull(), ULe8(108))
16     :AtLeast(1)
17     :WillRepeatedly(Return(&m_payloadBuffer));
18
19 // Example (d)
20 expect command Timer.startOneShot(UGt32(1024));
21 expect command Timer.startOneShot(Any());
```

Listing 7: Examples of modifiers controlling number of times expectations are matched

While expectations, argument matchers, sequences, and modifiers are mechanisms for asserting that the tested module correctly uses the modules on which it depends, actions are required to make the interactions between the tested module and the dependencies bidirectional. Whenever an invocation of a mocked function is correctly matched to an expectation, an action associated with this expectation is invoked. A common action in the previous examples is the `Return` action, which makes the mocked function return the value passed as a parameter to the action. In particular, if no action is specified for a function's expectation, `Return` with a default parameter is associated with this expectation: `0` if the function's result is integer; `NULL` if it is pointer; `SUCCESS` if it is *`error_t`*; and a zeroed structure or union if such a type is returned by value.

Apart from `Return`, the framework provides a few actions for frequently used testing patterns. Many of these actions are inspired by the peculiarities of sensornet software, notably its low-level interactions with hardware. For example, TinyOS, but also other sensornet development environments [Akhmetshina et al. 2003; Dunkels et al. 2004], provide asynchronous (or split-phase) operations. A split-phase operation is initiated by a function, which returns immediately. However, when the operation completes, which is often signaled by a hardware interrupt, a special user-provided callback function is invoked to continue processing with the result of the operation available. The callback invocation is often done from a different context (e.g., thread) than the invocation that initiated the operation, hence the name split-phase. In TinyOS, functions that can initiate operations are called *commands*, callbacks are called *events*, and execution contexts correspond to *tasks*. For instance, Listing 8 presents a low-level TinyOS frame sending interface. A user invokes the `send` command, which initiates a frame transmission and returns immediately. When the frame has been physically transmitted, an interrupt is signaled. The interrupt handler creates a new task. When the task is finally executed by a scheduler, it invokes the callback for the `sendDone` event, thereby allowing the user to continue after the transmission.

There are at least three control flow paths that a module using the split-phase interface from Listing 8 should be prepared to handle. First, the `send` command may fail, meaning that no `sendDone`

```
1 interface BareSend {
2   command error_t send(message_t * msg);
3   event void sendDone(message_t * msg, error_t error);
4   command error_t cancel(message_t * msg);
5 }
```

Listing 8: Example of interface for split-phase operations (with callbacks)

event will occur. Second, the `send` command may succeed, but the `sendDone` event may later report a transmission error. Third, both `send` and `sendDone` may be successful. Testing whether a module correctly handles these control flow paths, especially those involving the `sendDone` event, would normally require a lot of code: initiating the sending operation, starting a task, and invoking the callback. For this reason, our framework introduces special actions and constructs, which reduce the amount of code to just a few lines. As an illustration, Listing 9 presents entire code that has to be written to trigger the second control flow path. The essence is using a composite action, `DoAll`, that contains two sub-actions: `Return` and `GenerateEvent`. `Return` will make the mocked `send` command return *`SUCCESS`*, thereby informing the tested module that a send operation has been initiated successfully, and the module should later expect the `sendDone` event. `GenerateEvent` will in turn start a task, which, when executed, will actually invoke the event with an appropriate error code, `ENOACK`. A special modifier, `PassArg`, is introduced to pass the pointer to a message buffer provided by the user as the first (zeroth) argument of the `send` command to the `sendDone` event, which is required by the specification of the `BareSend` interface.

```
1 expect command BareSend.send(NonNull(), UGt8(0))
2     :WillOnce(
3         DoAll(
4             GenerateEvent(BareSend.sendDone, PassArg(0), ENOACK),
5             Return(SUCCESS)
6         ));
7
8 call BareSend.send(&m_frameBuf, m_frameLen);
```

Listing 9: Entire code to test control flow path that involves callbacks and context switches

There are many other constructs that our framework provides for embedded mock modules. Table I contains selected examples. Moreover, as we explain next, the design of the framework facilitates adding new ones on demand. In fact, many of the constructs now available out of the box have been introduced based on our experiences with the framework.

Table I: Examples of new constructs related to expectations for embedded mock modules

Type	Examples	Description
argument matchers	`Any`, `NonNull`, `UEq16`, `UGt8`, `ULe32`, `Ptr`	Verify whether the runtime parameters of a mocked function match an expectation for that function.
actions	`Return`, `Invoke`, `DoAll`, `GenerateEvent`, `PassArg`	Perform various operations in response to an invocation of a mocked function that matches an expectation.
action setters	`WillOnce`, `WillRepeatedly`	Associate one or more actions with an expectation.
modifiers	`Exactly`, `AtLeast`, `AtMost`, `AnyNumber`	Control how many times an expectation can be matched.
sequences	`NewSequence`, `AddToSequence`	Enable imposing an order of expectations and their actions.

In summary, the goal of the constructs, and embedded mock modules in general, is to encapsulate versatile logic that can be configured with just a few lines of code, thereby making tests compact.

As an exercise, imagine how much code it would take to test all control flow paths in the previous examples of a start sequence and split-phase operation if mock modules were unavailable. Moreover, mock modules allow for testing the internal behavior of a function, which is a major improvement over frameworks that provide just assertion macros: by nature, such macros alone are capable of testing functions only end to end. Finally, containing both expectations and invocations of tested functions, tests utilizing mock modules are self-descriptive: the entire test scenario is in one file.

4.4. Embedded Mock Modules: Implementation

However, while convenient for test developers, the support for mock-based testing poses a number of challenges for framework implementers. A major difficulty comes from the fact that, due to resource constraints of sensor nodes, predominant sensornet programming languages do not natively provide features that are the foundation of mocking frameworks for traditional computer software [Google 2008; Faber 2008], that is, the dynamic features in object-oriented and functional languages.

To illustrate, suppose that a tested module, m_t, invokes a function, foo, from a module, m_d; m_t thus depends on m_d. In object-oriented software, m_t and m_d would correspond to objects, o_t and o_d respectively, hence the invocation would correspond to invoking method foo of object o_d by object o_t. Thanks to one of the foundations of object-oriented languages—polymorphism—object o_d can be dynamically substituted with another object, o_m, that implements the same interface containing method foo. Not only can the substitution happen *at run time*, but also *transparently* to o_t. In other words, polymorphism enables dynamic binding between various system components (objects). This is crucial, in particular, for object-oriented mock frameworks that dynamically substitute a dependency, o_d, of an object with a mock object, o_m, whose foo's function behavior is configured at run time via expectations. In contrast, for efficiency, the predominant languages for sensornets, namely C and nesC, promote static binding: no later than at compile time, is m_t's invocation of foo linked to the implementation of foo in m_d. Transparently substituting m_d with a dynamically configured embedded mock module, m_m, thus requires special measures in these languages.

As another example, consider the argument matchers for expectations, such as `Any`, `NonNull`, or `UEq8`. As we mention previously, an argument matcher is conceptually a function that takes the run-time value of a mocked function's argument and returns a boolean depending on whether the value is as expected. However, argument matchers do not map directly to C (or nesC) functions. While matchers such as `Any` or `NonNull` could in principle be represented as C functions, matchers such as `UEq8` require different treatment because they are configurable at run time with parameters. More specifically, one or more parameters can be provided to a matcher as a part of an expectation for a mocked function. Yet, these parameters are used later, when the function is invoked with the actual arguments. For example, in Listing 6, `UEq8(COEFF_A)` specifies `COEFF_A` as the parameter of matcher `UEq8` for the first argument of function `Sensor.calibrate`. However, only when the function is invoked, will the matcher check if the first argument passed to the function in the invocation is equal to `COEFF_A`. Note also the same matcher is configured with a different parameter (`COEFF_B`) for the second argument of the same function. Therefore, rather than corresponding to C functions, argument matchers resemble lazy functions in functional programming languages. A lazy function can be presented at run time with just some of its arguments, thereby yielding a new function that takes the remaining arguments. Only when all arguments are provided, is the function evaluated. Neither C nor nesC support lazy functions natively, though.

These are just some of the dynamic features that a typical mocking framework employs and that are not elements of the predominant programming languages for sensornet systems. One possible way to enable mock based testing of a system written in such a language could thus involve rewriting the system in a language that does support the dynamic features. However, doing so just for the sake of testability is usually out of the question, especially considering the existing code base, the overhead of such features, and the resource constraints of sensor nodes. Another solution could involve emulating the dynamic features with the available programming constructs. In particular, polymorphism and lazy functions can be emulated with a combination of C structures and calls by function pointers. However, without support from the compiler, such a solution would be inconve-

nient for programmers and would require redesigning and reimplementing significant portions of the system, not to mention an increase in resource consumption. This suggests that embedded mock modules require both compile- and run-time support.

Following this reasoning, the support for embedded mock modules in our framework comprises two elements: language extensions and a run-time engine. The language extensions encompass automatically generated mock modules and the constructs presented in the previous listings (i.e., expectations); they can be used only from within test code. The run-time engine, in turn, emulates all the dynamic features necessary for the generated code and these constructs. The interplay between the language extensions and the run-time engine is illustrated in Fig. 4: a modified nesC compiler generates code for mock modules and the extensions, and the code contains mainly invocations of well-defined functions of the run-time engine. Let us discuss these ideas in more detail.

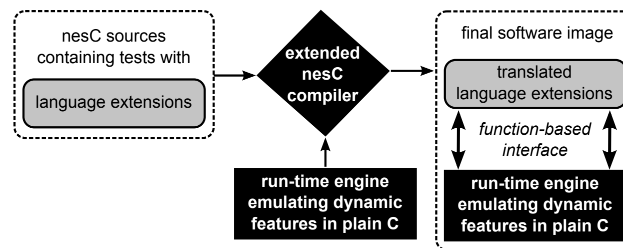

Fig. 4: Overview of implementation of embedded mock modules

Recall that a test developer creates or updates a test suite for a module with a command line tool. One of the tasks of the tool is identifying all interfaces the tested module uses. For each such interface, the tool generates a skeleton of an embedded mock module file that can later be processed by the extended compiler (cf. Listing 10). Moreover, it creates linker rules (a makefile) that assemble the tested module, the embedded mock modules, the run-time engine, and other run-time components of the framework and TinyOS into a single code image that can be installed on a node or run in a simulator. In essence, the rules isolate the tested module from the rest of the system by statically binding all interfaces the tested module uses to provide its functionality to the embedded mock modules rather than to the real modules on which the tested one depends. With this solution, no existing module code has to be altered to enable mock-based unit tests.

```
1  mock module BareSendM {
2    provides interface BareSend;
3  }
```

Listing 10: Example of automatically generated skeleton of embedded mock module

Upon compilation, when the extended nesC compiler encounters a skeleton mock module, it generates an implementation for all the functions of the interface the module provides and allocates structures for the internal module state. Each generated function can be invoked by the tested module in different test scenarios and may be required to behave differently in these scenarios, depending on the expectations set for this function in a given scenario. Due to the static binding between the tested module and the function, substituting the function's body at run time is impractical. Consequently, instead, the expectations for the function are stored in the module's state, and are used to alter the run-time behavior of the function. From a high-level perspective, the body of each automatically generated function essentially consists of scanning the expectations for the function to identify one that matches the values of the function's arguments and executing actions associated with this expectation. The details are somewhat more intricate, because expectations can be accompanied

by various modifiers, can belong to sequences, may not match at all or may have been matched previously, in which case appropriate errors should be reported, to name just a few reasons.

An important aspect of the generated code is its size. A mocked interface may consist of multiple functions. Moreover, the tested module may use multiple interfaces. As a result, without special measures, the code that would have to be generated for all the mocked functions in a test suite could be sizable, which might be problematic for some sensor node platforms. For this reason, our framework strives to minimize the amount of the generated code by making use of the run-time engine. Without getting into details, since the previous algorithm for a mocked function is largely generic, we provide just a single generic implementation instance of it. This instance is a part of the run-time engine and is shared by all mocked functions. With this solution, the code generated for each function boils down to transforming aspects specific to this particular function into generic constructs that can be fed to the algorithm in the run-time engine. In other words, the type-specific arguments and return value of the function are wrapped into generic emulated objects, which the algorithm within the run-time engine can handle. In effect, the code generated for each mocked function is just several memory copy instructions and run-time engine function invocations.

Similarly, the new language constructs for setting expectations are translated into run-time engine function invocations. The invocations resulting from an expectation essentially change the state of the mock module for which the expectation is set. Their number may vary, depending on the number of argument matchers, actions, modifiers, and sequences specified in the expectation. We omit the details for brevity.

As to the run-time engine itself, it implements the functions for configuring expectations, and the algorithm for matching them and triggering their associated actions. It does so by internally emulating polymorphic objects by means of C structures and function pointers. To illustrate, let us present the entire process of adding a new argument matcher, `UBetween32`, to the framework: a matcher checking that an unsigned 32-bit argument of a mocked function is between a minimal and maximal value. Listing 11 presents a sample use of the matcher in an expectation.

```
1  expect command Timer.startOneShot(UBetween32(MIN_BACKOFF / 2, MAX_BACKOFF));
```

Listing 11: Sample use of new matcher in expectation

Recall that a matcher is conceptually a lazy function. For this reason, internally, the run-time engine represents a matcher as a C structure whose first field is a pointer to the actual matching function (cf. Listing 12). The structure can be thought of as the root of a polymorphic “class” hierarchy representing various matchers. Any matcher invoked from within the framework is casted to this structure and the function represented by the first field of the structure is invoked by pointer. As a side note, the example also gives a hint of why mock-related constructs require framework support to be convenient for programmers and how much the framework actually automates.

```
1  // Structure representing lazy matcher function.
2  typedef struct BaseMatcher {
3    uint8_t (*match)(void *, void *);
4  } BaseMatcher;
5
6  // Use of the function inside the run-time engine.
7  BaseMatcher * m = ...;
8  void * wrappedFunctionArg = ...;
9  bool matches = m->match(m, wrappedFunctionArg);
```

Listing 12: Abstract matcher as emulated polymorphic object representing lazy function

To implement our matcher, `UBetween32`, three elements are necessary, as shown in Listing 13. First, a matcher structure, `UBetween32Matcher`, represents the lazy matcher function with space

for the two parameters, the minimal and maximal value. It is vital that the first field of the structure is the same as in `BaseMatcher`, which allows for the aforementioned casting. Second, function `UBetween32Matcher_match` is the implementation of the matching function. It takes the matcher structure and the actual argument value of the mocked function that has to be matched. As mentioned previously, the argument value is wrapped into a generic object. Therefore, before it can be compared with the matcher's minimal and maximal values, it has to be unwrapped. The function returns a boolean denoting the result of the comparison. Finally, function `UBetween32` is a constructor that creates the matcher structure and stores the two parameters in the appropriate fields. As such, the constructor is precisely the narrow interface to the matcher that test developers use in expectations (cf. Listing 11).

```
1  // Structure representing lazy matcher function.
2  typedef struct UBetween32Matcher {
3    uint8_t (*match)(void *, void *);
4    uint32_t minVal;
5    uint32_t maxVal;
6  } UBetween32Matcher;
7
8  // Actual matcher function.
9  uint8_t UBetween32Matcher_match(void * mp, void * wrappedFunctionArg) {
10   UBetween32Matcher * m = (UBetween32Matcher *)mp;
11   uint32_t actualFunctionArg = tm_unwrapUInt32(wrappedFunctionArg);
12   return actualFunctionArg >= m->minVal && actualFunctionArg <= m->maxVal;
13 }
14
15 // Constructor (used in expectations).
16 BaseMatcher * UBetween32(uint32_t minVal, uint32_t maxVal) {
17   UBetween32Matcher * m =
18       (UBetween32Matcher *)tm_palloc(sizeof(UBetween32Matcher));
19   m->match = &UBetween32Matcher_match;
20   m->minVal = minVal;
21   m->maxVal = maxVal;
22   return (BaseMatcher *)m;
23 }
```

Listing 13: Complete implementation of sample new matcher from Listing 11

By and large, creating new matchers (and other constructs in the run-time engine) may involve a few lines of pointer-juggling code, but the scheme is fairly straightforward once understood. Moreover, using the new constructs requires hardly any effort. This degree of openness is an important property of the run-time engine and embedded mock modules in general, because many of the constructs that are now provided out of the box emerged from our experiences with the framework, and it is likely that future users will create even more constructs that best suit their testing patterns.

To sum up, the implementation of embedded mock modules consists of the compile-time language extensions and the run-time engine. The language extensions are convenient means for succinctly expressing complex albeit common testing patterns. The run-time engine, in turn, encapsulates the algorithms and constructs necessary to dynamically mock embedded code, and allows for defining new constructs. Such a solution is motivated, on the one hand, by the convenience of test developers, and on the other, by the resource constraints of sensor nodes and the related limitations of predominant sensornet programming languages. With this solution, no changes to the existing module code are necessary to make it unit-testable with embedded mock modules, hence adopting unit testing need not lead to performance degradation or increase in resource usage of the code.

5. EXECUTION PERSPECTIVE

While embedded mock modules minimize the overhead incurred by developing unit tests, similar solutions are necessary to minimize the effort involved in executing the tests. Drawing from a decade of experience with sensornet testbeds [Ertin et al. 2006; Werner-Allen et al. 2005], our framework provides mechanisms that, triggered by a single command, compile a test suite and install and run it on a node or in the TinyOS simulator, TOSSIM [Levis et al. 2003], thereby producing diagnostic output that can be interpreted by developers. These mechanisms are thus fairly standard.

In contrast, a fundamental problem with test execution is that both the tested code and its test logic may have defects, which manifest as failures during the execution. Our framework strives to handle many different types of failures and to provide relevant diagnostic information. In effect, a single test run is often sufficient to identify a failure and pinpoint its root cause, and rarely is the developer forced to employ a hardware debugger. Such a solution is crucial especially for interactive software development methodologies, such as TDD, in which unit tests are initially *expected to fail*, and the developers must have means for identifying and fixing the causes of the failures on the fly. At the same time, however, detecting and intercepting failures is not trivial in sensornets, because sensor nodes lack the hardware and operating system support that is traditionally employed for containing failures (e.g., privileged instructions, memory protection, exceptions).

Yet, compared to existing frameworks, our framework has an important advantage in containing faults, that is, the isolation embedded mock modules provide for the tested module from other modules, but also from hardware. Linking the interfaces used by the tested module to mock modules rather than to real modules minimizes the amount of software that can potentially fail during test execution: provided that the framework code is correct, potential flaws may exist only in the tested module or the test logic itself. By additionally requiring that, rather than directly accessing I/O pins, special registers, etc., the tested module refers to these low-level hardware components through interfaces, the tested module can also be isolated from hardware, thereby yielding full control over the hardware to the framework. Counterintuitively, such hardware isolation does not preclude testing device drivers. This is because sensornet operating systems typically expose hardware via a thin, “dumb” presentation layer, usually inlined during compilation [Levis et al. 2005]. In contrast, the actual driver logic, which is often complex, is implemented in higher-layer modules that use these interfaces. In fact, not abstracting hardware is a type of software flaw our framework aims to prevent. Therefore, the only low-level code the framework is unable to test due to the isolation is a thin hardware presentation layer. We believe that this is acceptable, considering that, in the absence of the isolation, tracking many types of faults without a debugger would be a daunting task.

Apart from the isolation, another advantage of embedded mock modules is that they facilitate catching many classes of failures early during execution. More specifically, mock modules facilitate testing the interactions of a function with other functions. In particular, if the function invokes those functions an unexpected number of times, in a wrong order, or with invalid arguments, an appropriate mock module immediately intercepts and reports the failure. In other words, apart from catching failures only end to end (e.g., by checking a result of a function), mock modules also detect problems in the internal behavior of functions. Combined with problems that can be detected by end-to-end assertions, problems detectable by mock modules thus already cover most classes of software defects. As a result, we provision mechanisms for just two more types of defects.

First, a test case may block. One sample cause may be that, under some condition produced by the test case, the tested module or the test logic itself enters an endless loop. A more elaborate one is an invalid control path, for instance, in a corner case of the module or the logic, that lacks an invocation of a callback completing an asynchronous operation. Our framework addresses this issue with a dedicated hardware timer, which is restarted for each test case. An interrupt from the timer indicates that the current test case has timed out. Since an interrupt preempts the regular-mode execution of the tested code, a timed-out test case can always be stopped. In addition, the user is given control over the value of the timeout, so that it can be adjusted to different test cases.

Second, a test case may also corrupt node memory. Without memory protection, as is the case for sensor nodes, an invalid pointer or out-of-bounds array access may arbitrarily disrupt the node's state, including the internal components of the framework. To alleviate such problems, the framework relies on the memory safety extensions for TinyOS [Cooprider et al. 2007]. The extensions comprise annotations and a toolchain based on the Deputy compiler [Condit et al. 2007] for the C language. The annotations provide additional semantic information for potentially dangerous memory accesses. This information is translated at compile time into code that checks the correctness of these accesses. Therefore, if at run time a potentially dangerous memory access fails to pass the associated check, it can be intercepted before it corrupts memory. Such an approach allows for detecting most types of memory access failures, but requires using the safety extensions in the module code and itegrating them with our framework. Moreover, the internal framework constructs must be trusted to operate correctly, as they do not use the safety extensions.

Being able to detect and intercept failures still requires means for reliably reporting the failures to the user. The reporting is realized by means of serial or radio communication. Upon a failure, the node sends an appropriate diagnostic message to a frontend application running on the user's PC. The application displays the received message on a console, so that it can be interpreted by the user. While seemingly straightforward, this scheme also necessitates special measures.

More specifically, when a failure is intercepted by the failure handler within the framework, the execution of the faulty code is stopped only for the period of executing the handler. Therefore, if the handler returned, the execution of the faulty code would continue where it stopped, which could lead to an arbitrary behavior and possible subsequent failures. This suggests that the execution of the test code should be stopped permanently upon a failure. This is not trivial, though, because the faulty code may have started threads (tasks in the TinyOS nomenclature) that will continue executing despite stopping the execution of the failing thread. On the other hand, stopping all threads of the node would essentially preclude reporting the failure, as the framework and operating system use threads internally to implement the communication services. We address this problem by making all error reporting operations blocking from the perspective of the tested code.

A simple solution is to introduce a blocking version of the embedded `printf` function, so that after the function finishes execution, we are guaranteed that the diagnostic information has actually been sent to the PC. With such guarantees, a failure handler comprises invoking the `printf` function with an appropriate error message and resetting or stopping the entire node, so that no subsequent failures resulting from the intercepted one are reported. This solution works well for serial communication, which typically requires no more than a few lines of code to provide the blocking embedded `printf` function, and for TOSSIM, which supports `printf` natively.

A more general solution, in turn, comprises two mechanisms: a custom task scheduler and exceptions. The main idea behind the scheduler is to provide two classes of tasks: tasks spawned by the tested code and the mock modules and tasks spawned by the operating system and framework communication services. Tasks from the first class have a lower priority than tasks from the second class: if there is any active task from the second class, it is always executed before any task from the first class. The main purpose of an exception is in turn stopping the current execution context and transferring control to the scheduling routine for the tasks of the second class, such that no tasks from the first class will ever be executed again. The implementation of exceptions is based on the `setjmp` and `longjmp` standard C functions, which makes it portable. An error handler thus invokes the embedded `printf` function, which may now be nonblocking, and raises an exception. The exception guarantees that no test code is ever executed again; the scheduler guarantees that any communication resulting from invoking `printf` can be completed irrespective of whether the communication is blocking or relies on spawning tasks. Such a solution works with serial and radio communication, and even allows for implementing `printf` on top of network protocols, such as Collection Tree Protocol (CTP) [Gnawali et al. 2009]. However, it requires changes in the communication drivers and services to make them explicitly spawn tasks from the second class. Therefore, it is not fully supported at the time of writing.

There are several other run-time issues the framework addresses, like arbitrating access to the tested module or seamlessly supporting the same model on nodes and in TOSSIM. It also leverages mechanisms provided out-of-the-box by nesC, such as race-condition detection [Gay et al. 2003]. Since, in contrast to the previous ones, many of those issues are specific to TinyOS, we omit them for brevity. By and large, the aforementioned solutions strive to detect, intercept, and report many classes of failures, so that cases when a debugger is necessary to diagnose flaws in the code are infrequent.

6. ASSESSMENT PERSPECTIVE

Being able to develop and execute unit tests, one still needs solutions for assessing how well the tests exercise code. Modern practices suggest employing code coverage tools to this end, like gcov [Gough 2004] for GCC or Cobertura [Doliner 2006] for Java. Using such tools boils down to compiling source code and running the resulting binary, which produces a report that contains the original source code annotated depending on the run-time control flow. The annotations allow for precisely identifying which pieces of the code have not been executed. Since such information is invaluable for assessing the quality of tests, we implemented code coverage mechanisms in our framework.

6.1. Code Coverage

A popular code coverage metric is line coverage, which, for each line, specifies whether that line is executed during a test. Figure 5 presents a fragment of a sample line coverage report obtained with our framework for the TinyOS driver of the CC2420 radio chip (file `CC2420ReceiveP.nc`). Each line annotated with “`1`” was executed at least once. Lines annotated with “`#####`” were not executed at all. Finally, lines annotated with “`-`” were not considered for execution. In particular, the report shows that during the test the driver assumed no hardware address recognition (line 830 was not executed), hence another test may be necessary. Likewise, no radio frames with extended IEEE 802.15.4 addresses were received (lines 837-839 were not executed), which suggests that yet another test may be a good idea to thoroughly exercise the driver.

```
    1:  829:    if(!(call CC2420Config.isAddressRecognitionEnabled())) {
#####:  830:      return TRUE;
    -:  831:    }
    -:  832:
    1:  833:    if (mode == IEEE154_ADDR_SHORT) {
    1:  834:      return (header->dest == call CC2420Config.getShortAddr()
    -:  835:              || header->dest == IEEE154_BROADCAST_ADDR);
    1:  836:    } else if (mode == IEEE154_ADDR_EXT) {
#####:  837:      ieee_eui64_t local_addr = (call CC2420Config.getExtAddr());
#####:  838:      ext_addr = TCAST(ieee_eui64_t* ONE, &header->dest);
#####:  839:      return (memcmp(ext_addr->data, local_addr.data, IEEE_EUI64_LENGTH) == 0);
    -:  840:    } else {
    -:  841:      /* reject frames with either no address or invalid type */
    1:  842:      return FALSE;
    -:  843:    }
```

Fig. 5: Fragment of sample line coverage report generated by our framework

Another coverage metric the framework provides is branch coverage, which, for each conditional instruction, explains why the instruction is executed in a test. Branch coverage is particularly useful for complex conditional instructions. To illustrate, Fig. 6 presents a fragment of a sample report obtained for the TinyOS route maintenance module (file `CtpRoutingEngineP.nc`) of the Collection Tree Protocol (CTP) [Gnawali et al. 2009]. The presented code is a conditional instruction with an alternative of two basic conditions, so-called *atoms*: `entry->info.parent == INVALID_ADDR` and `entry->info.parent == my_ll_addr`. The corresponding line coverage report would annotate all lines as executed during the test. However, without branch coverage, we would not know why the lines inside the conditional instruction were executed: was it

because `entry->info.parent == INVALID_ADDR` or because `entry->info.parent == my_ll_addr` in some control flow? The branch coverage report provides the explanation.

```
            if (entry->info.parent == INVALID_ADDR || entry->info.parent == my_ll_addr) {
####--,-+
              dbg("TreeRouting",
                  "routingTable[%d]: neighbor: [id: %d parent: %d  etx: NO ROUTE]\n",
                  i, entry->neighbor, entry->info.parent);
              continue;
            }
```

Fig. 6: Fragment of sample branch coverage report generated by our framework

There were at least two different control flows in the depicted code. In one of the flows, annotated by “−−”, neither of the two atoms was true, hence the two “−”. However, in another flow, annotated by “−+”, the first atom was false, but the second was true. In other words, the body of the conditional instruction was executed, because in some control flow during the test `entry->info.parent == my_ll_addr`. In contrast, the path with `entry->info.parent == INVALID_ADDR` was not exercised in any flow of the test, hence another test case may be necessary to fully exercise the code. More complex conditional instructions with many conjunctions and alternatives are equally easy to analyze given a branch coverage report.

6.2. Initial Implementation Approach

Generating the aforementioned coverage reports requires compile- and run-time support for tracking which pieces of code are executed. Many sensornet programming environments (including TinyOS) rely on GCC to produce the final binary software images for sensor nodes. Since gcov is part of the GCC toolkit, employing it to enable code coverage would be a natural approach.

Gcov requires passing two options to GCC when producing the software image for sensor nodes: `-fprofile-arcs` and `-ftest-coverage`. They inform GCC that the generated image should be instrumented as follows. Before being converted into target microcontroller instructions, the software image is represented as a directed graph of so-called basic blocks. A basic block is a maximal contiguous piece of microcontroller-oblivious, intermediate code, such that a jump out of the block occurs only at the end of the block while any jumps into the block can lead only to the beginning of the block. In other words, the code in a block is always executed entirely, and the jumps—the edges in the block graph—reflect the possible control flow. The `-fprofile-arcs` option allocates in the final image a 64-bit counter for every edge in the basic block graph and injects instructions that increment the counter on the jump corresponding to this edge. The `-ftest-coverage` option, in turn, has two roles. First, it stores the basic block graph into auxiliary files. Second, it links the image with code dumping the counter values at runtime to another file. What gcov does is merge the basic block graph file, obtained during compilation, with the counter values file, obtained by running the image, into a human-readable coverage report.

While seemingly ideal for sensornets, gcov fails in practice. To start with, GCC is not the only C compiler for sensornets, and the aforementioned instrumentation is not supported by many other embedded C compilers; neither is it supported by all GCC platforms. More importantly, even on the supported platforms, the instrumentation works poorly under the constrained resources of a sensor node. The 64-bit counters consume lots of precious RAM. Likewise the code incrementing them on jumps is large, as it uses merely 8- or 16-bit instructions. Yet, the counter width cannot be simply reduced, because, apart from gcov, the very same instrumentation is used, for instance, by profiling tools, like gprof, where the large width is important. Furthermore, the instrumentation unit is C file, meaning that it is impossible to select at compile time which jumps are instrumented. This is problematic for sensornet systems that often group multiple files into a single compilation unit to enable aggressive optimizations. An extreme case is nesC, which produces a single C file containing the entire node image. As we show in Section 8, the counter width and instrumentation granularity

make it virtually impossible to test coverage of even simple sensornet applications. Finally, there is no standardized way for accessing the counters at a node, for example, to send them to a frontend application running on the user's PC. The `-ftest-coverage` option simply assumes a classical file system, which is not available on many node platforms.

6.3. Final Implementation

Although we could modify the open-source GCC to address these issues, this would introduce severe maintenance and portability problems. For this reason, rather than patching GCC and adopting gcov, we provide novel, practical coverage mechanisms, customized for resource-constrained devices.

To maximize portability, the mechanisms instrument code *before* it is passed to a C compiler. More specifically, we have extended the nesC compiler such that the plain C code it produces from nesC source files is already instrumented. Figure 7 illustrates these principles. First, nesC source code is instrumented by the extended nesC compiler. The resulting C file is compiled normally (by default, by GCC) into a binary software image. The image is then installed and executed on a node or in TOSSIM. Finally, after the execution, a custom `nesccov` script produces human-readable coverage reports for each of the original nesC source files. A similar solution could be implemented for programming environments based on plain C rather than nesC. The role of the extended nesC compiler in instrumenting the source code could be taken over by a custom C-to-C translator, for example, one based on CIL [Necula et al. 2002].

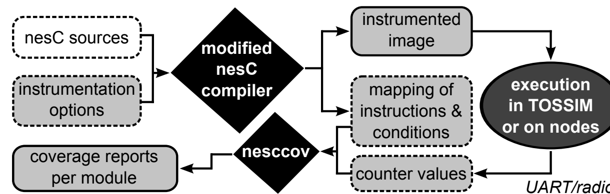

Fig. 7: Overview of implementation of code coverage mechanisms

Since the instrumentation is just plain C code without any low-level hardware-specific instructions, the solution is highly portable. Moreover, since we are in full control of the instrumentation, we can also control its overhead. In particular, rather than forcing instrumentation of entire C files, we provide a command-line option to the extended nesC compiler that takes a regular expression specifying the names of the individual functions to be instrumented. Another option enables in turn changing the width of counters used for instrumentation, even down to just one bit per counter, which is sufficient for code coverage. Finally, additional options select the type of instrumentation (i.e., line and/or branch) or prevent instrumenting time-critical code executed with disabled interrupts or entire code directly reachable from interrupt handlers. All in all, the users are offered a rich set of mechanisms for reducing the instrumentation overhead depending on their testing policies.

However, implementing the instrumentation at the high, source code level rather than at the lower, intermediate code level is problematic. In particular, at the high level, there is no flat graph of basic blocks, just a hierarchical abstract syntax tree that reflects the language constructs in the source code. Operating at a different level thus requires revisiting the line and branch coverage algorithms.

When it comes to line coverage, a naïve solution could involve associating a counter with every C instruction and generating an instruction incrementing the counter before the instrumented instruction itself. However, this solution would lead to an explosion of the counter tables, effectively precluding practical applications. Therefore, instead, source code lines always executed together must be grouped into chunks, much like basic blocks, but at the abstract syntax tree level. To this end, the abstract syntax tree is traversed to identify subtrees representing such chunks. For each

such subtree, a static counter is allocated in the output C file, and the subtree is extended with an instruction incrementing the counter. In the basic block graph, this would correspond to instrumenting vertices rather than edges, which further saves RAM.

For branch coverage, in turn, only subtrees representing boolean expressions, like in conditional instructions and loops, are identified in the abstract syntax tree. Given such a subtree, a counter is allocated for every possible evaluation of the corresponding expression. The subtree is also extended with instructions incrementing appropriate counters, depending on the runtime evaluations of the expression. For example, in expression `a||b||c`, there are three atoms, `a`, `b`, and `c`, each of which can evaluate to true or false. For an unspecified evaluation order, the expression has $2^3 = 8$ possible evaluations, hence 8 counters are allocated. We have also implemented algorithms reducing the number of counters by assuming lazy evaluation, but we omit them here for brevity.

To sum up, code coverage is popular for assessing the quality of tests. Measuring code coverage, however, requires instrumenting the tested software. Due to the resource constraints of sensor nodes, such instrumentation necessitates dedicated solutions for sensornets. In contrast to traditional ones, the presented solutions operate at the abstract syntax tree level. Not only does this facilitate controlling the overhead of instrumentation, but also ensures portability. In effect, as we confirm empirically in Section 8, the solutions enable measuring coverage of even large sensornet systems.

7. MANAGEMENT PERSPECTIVE

The features of our framework discussed hitherto are *mechanisms* supporting unit test development, execution, and assessment. In contrast, test management is concerned mostly with defining and enforcing appropriate *policies*. The definitions of the policies depend on a particular software development organization and methodologies employed in that organization. Enforcing the policies, in turn, is typically realized with traditional mechanisms, such as test databases, code review software, and, in general, development support software. No special test management mechanisms are required for sensornet software. Consequently, the framework provides no additional dedicated management mechanisms, and instead relies on integrating it with the mechanisms an organization employs daily.

However, the mechanisms provided by the framework for test development, execution, and management allow for effectively enforcing new policies. For example, as embedded mock modules greatly reduce the effort involved in developing unit tests, an organization may introduce a policy requiring each module to be accompanied by such tests. To enforce this policy, whenever code is committed to the code repository, execution of relevant test suites may be triggered automatically with the code coverage measurement active. If too much of the new code is uncovered, the code may be automatically rejected, and its author may be provided with feedback information. Likewise, if any of the test fails, the code may be rejected with appropriate feedback to the author. Although such policies are common in organizations developing reliable distributed systems, to date, adopting them in sensornet communities has been largely hindered by the lack of the appropriate mechanisms.

8. EVALUATION

To date, evaluations of unit testing frameworks, if conducted at all, have focused on qualitative rather than quantitative aspects [Moss 2007; Okola and Whitehouse 2010]. In contrast, apart from demonstrating qualitative benefits, we aim to show as many quantitative ones as possible, because we hope that they will stimulate adoption of our framework. To the best of our knowledge, this is the first such an evaluation of unit testing in sensornets.

Since our framework is implemented in TinyOS, the evaluation inherently targets the TinyOS code base. Nevertheless, as most of the problems the framework addresses are fundamental, we believe that similar results could be obtained for other sensornet programming environments.

8.1. Module Dependency Analysis

Let us start by demonstrating that module dependencies are a major obstacle for adopting unit testing in sensornets. In effect, without embedded mock modules, as provided by our framework, testing an average module in isolation is bound to be costly.

To perform the dependency analysis, we have extended our toolkit with a meter suite for nesC software. We subsequently ran the suite on a recent revision (6001) of the TinyOS sources. Table II presents the resulting statistics relevant to module dependencies.

Table II: Overview of module dependencies in TinyOS

GENERAL INFORMATION	
#interfaces	661
#configurations	1008
#modules	981

INTERFACES	
avg. #commands	4.80
avg. #events	0.97

MODULES			
avg. #used ifaces	4.46	avg. sum #cmds in used ifaces	23.57
		avg. sum #evts in used ifaces	3.05
avg. #provided ifaces	2.12	avg. sum #cmds in provided ifaces	6.97
		avg. sum #evts in provided ifaces	1.57

The TinyOS code base encompasses 981 modules that interact through subsets of 661 interfaces. An average interface consists of nearly 5 commands and 1 event. An average module uses more than 4 and provides more than 2 interfaces. Therefore, to test a module in isolation from other modules, more than 4 interfaces have to be abstracted out on average. Since abstracting an interface requires implementing all its commands, a rough calculation shows that abstracting out dependencies of an average module requires implementing more than 20 command stubs (functions). As some interfaces are more popular than others, the actual number of functions that have to be implemented to abstract out an average module is even higher: 23.57. With such a large amount of work necessary to isolate a single module, unit testing without mock modules becomes extremely laborious.

Figure 8 confirms that this result is not just due to a few modules having an excessive number of dependencies. More than a half of the modules depend on at least 3 interfaces, while only fewer than 15% have no dependencies. Likewise, isolating more than a half of the modules requires implementing at least 15 functions. Importantly, modules that have the most dependencies also implement a lot of logic that requires thorough testing. In particular, both `CC2520DriverLayerP`, which uses the most interfaces, and `Msp430Adc12ImplP`, whose dependencies involve the most functions, implement complex device driver logic (above the hardware presentation layer). The cost of testing this logic in isolation is simply prohibitive without embedded mock modules.

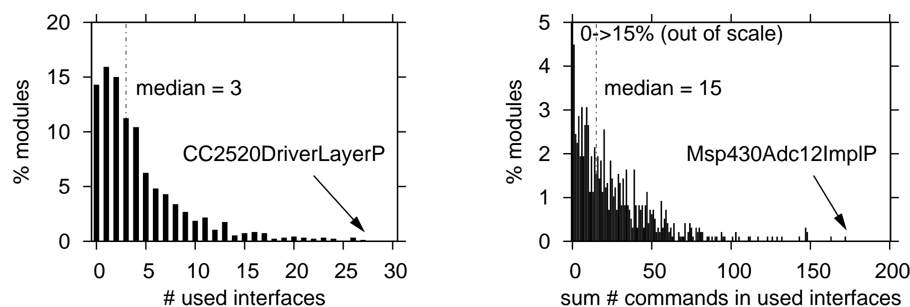

Fig. 8: Distribution of module dependencies in TinyOS

Although one could imagine implementing interface stubs that can be shared among unit tests, the 661 interfaces yield a total of 3170 command stubs to be implemented. Moreover, such shared stubs would have to provide additional interfaces for configuring their behavior from within test logic. Manually implementing stubs is thus uneconomic, thereby precluding testing the modules in isolation without the automatically-provided embedded mock modules.

8.2. Test Coverage Analysis

Since manually abstracting out module dependencies is bound to be costly, it may be argued that instead of testing the modules in isolation, one may combine them into relatively self-contained subsystems and test those instead. This approach is followed in the TinyOS code base, which does contain some tests of entire subsystems. However, although such integration tests are necessary to test how the modules interact with each other and with hardware, they cannot replace unit tests. In particular, past experiences suggest that integration tests fail to exercise all control flow paths and corner cases of the tested modules [Barrenetxea et al. 2008; Langendoen et al. 2006; Ramanathan et al. 2005; Tolle and Culler 2005], thereby increasing the likelihood of hidden software flaws.

Armed with our embedded code coverage solutions, we are finally able to verify this hypothesis. To this end, we have instrumented the applications from the TinyOS `apps` directory and its subdirectories (i.e., the aforementioned integration tests) and all the TinyOS modules these applications utilize (e.g., hardware drivers, protocols, system component, libraries). We managed to instrument the following applications: `AdcSimple`, `Blink`, `LplBroadcastCountToLeds`, `LplBroadcastPeriodicDelivery`, `LplUnicastPeriodicDelivery`, `MultihopOscilloscope`, `MultiLedSingle`, `Oscilloscope`, `RadioCountToLeds`, `RadioSenseToLeds`, `RadioStress`, `RssiToSerial`, `Sense`, `TestAcks`, `TestAdc`, `TestAM`, `TestDissemination`, `TestFcfsArbiter`, `TestLpl`, `TestNetwork`, `TestNetworkLpl`, `TestPacketLink`, `TestPowerManager`, `TestPowerup`, `TestRoundRobinArbiter`, `TestScheduler`, `TestSrp`, `TestTimerSync`. The rest is unmaintained or failed to compile out-of-the-box for the TelosB platform we had access to.

We then ran the instrumented code on our medium-scale testbed [Iwanicki et al. 2008]. The testbed consisted of 55 TelosB nodes dispersed in 5 office rooms. For the employed radio transmission power, the nodes formed a network with a diameter of 5 hops, a nonuniform density ranging from 8 to 31 neighbors per node, and many asymmetric links. All the applications were tested in their default TelosB configurations for at least 15 minutes each, except for `TestNetwork` and `TestNetworkLpl` that are more complex, incorporating among others the Collection Tree Protocol (CTP) [Gnawali et al. 2009] and the Four-Bit Wireless Link Estimator (4bitLE) [Fonseca et al. 2007], and were thus run for 24 hours each. In addition, for these two applications, we varied the location of the sink to be on the edges and in the center of the testbed. This, combined with the properties of the testbed, notably regions of high density and asymmetric links, and environmental impact resulting in connectivity changes, increased the chances of triggering corner cases in those applications. All in all, we believe that the setup should produce reasonable coverage numbers.

Nevertheless, Table III confirms the earlier hypothesis that integration tests alone are insufficient to achieve full coverage. In fact, the achievable coverage is considerably below what one can encounter in quality-assurance requirements of companies concerned with reliable systems. In an average module, only 72% lines of code are executed and only 57% branches are triggered.

Table III: Coverage of existing TinyOS integration tests

LINE COVERAGE	
#modules w/ $>$0% coverage	85
sum #lines in modules	5187
sum #covered lines	3918
median module coverage	82%
avg. module coverage	72%

BRANCH COVERAGE	
#modules w/ $>$0 cond. instr.	65
sum #branches in modules	1374
sum #covered branches	836
median module coverage	61%
avg. module coverage	57%

Moreover, note that Table III considers only modules in which at least one line or condition, respectively, is executed, and that the coverage for each such module is a union of the coverage from individual test applications. However, many modules are not covered by *any* standard test, hence taking them into account would further decrease the average coverage. Likewise, even tests dedicated for particular modules have a low coverage. For example, the `TestNetwork` application, which aims to exercise CTP, for the core `CtpRoutingEngineP` and `CtpForwardingEngineP` modules achieves only 77% and 69% line coverage and 69% and 68% branch coverage, respectively. These results confirm that a large number of control flow paths are indeed hardly ever exercised by integration tests. Consequently, any potential defects they contain may not be detected during testing, as suggested by past deployments.

The experiments also confirm that obtaining the code coverage data would be impossible without our dedicated solutions. To illustrate, Table IV compares the memory overhead of our instrumentation (all modules, one bit per counter) and gcov. Gcov's overheads, notably on RAM, are so large that only simple test applications that use few hardware components can be instrumented. In particular, with gcov, we have not managed to compile any test that exercises the simplest TelosB radio stack beyond periodically broadcasting a packet, `TestAM`. In contrast, with our mechanisms, the overheads are small enough to enable instrumenting all modules of even large integration tests. Moreover, for small tests, like those that gcov is also able to instrument, the most overhead is due to the extra code for reporting counter values to a PC rather than due to the instrumentation itself. Our solutions require less memory, because their counters are narrower (1 vs. 64 bits) and there are fewer of them. More importantly, however, the solutions enable precisely selecting the code instrumented for a test, which is crucial for resource-constrained devices.

Table IV: Memory overhead due to instrumentation

INSTRUMENTATION FOR TELOSB	RELATIVE AVG. MEMORY GROWTH	
	ROM OVERHEAD	RAM OVERHEAD
gcov ∗ §	204% / ? †	3,362% / ? †
our line	118% / 75% †	246% / 63% †
our branch	133% / 100% †	196% / 69% †
our line+branch	188% / 145% †	275% / 135% †

∗ Only 11 applications compile without exceeding the MCU memory limit.
§ Does not include code for reporting counter values by the node.
† Avg. over the 11 small apps that compile for gcov / avg. over all apps.

We have conducted other experiments, in particular, for the runtime overhead of the instrumentation. In an average, uninstrumented, non-stress-test application, the MCU sleeps 96% of the time. With our instrumentation (line and branch coverage), this value drops to 89%. The increase is not drastic, meaning that it does not significantly change the MCU utilization characteristics, which could in turn affect the coverage. Moreover, the runtime overhead, like memory, can be minimized further with the options our solutions offer. For example, uninteresting modules and time-sensitive interrupt code can be excluded from instrumentation. In any case, for development or pre-deployment testing, the small increase in runtime overhead is by and large irrelevant. What is important, in turn, is that our coverage solutions precisely identify untested control flow paths, thereby automating test assessment.

8.3. Benefits of Framework-assisted Unit Testing

Hitherto, we have quantitatively shown that, even when only intermodule dependencies are considered, unit testing without support from a framework is bound to be costly, and that integration testing does not sufficiently cover the tested code, thereby increasing the probability of undetected software flaws. Demonstrating the benefits of framework-assisted unit testing in a similar, quantitative manner is far more difficult, though, which is likely why it is rarely done. Therefore, rather than the ultimate evaluation of our framework, we present preliminary small-scale studies that just aim to

illustrate the most important advantages the framework offers. Nevertheless, to maximize interest in our work, apart from qualitative results, we strive to provide as many quantitative ones as possible.

The studies involved three developer groups. The first group consisted of two graduate students implementing a research prototype of an aggregation protocol for mobile networks [Gregorczyk et al. 2014], employing a new link-layer primitive [Pazurkiewicz et al. 2014]. The students initially lacked any TinyOS experience. The second group consisted of four people building a new sensornet hardware-software platform, two of which had had no prior TinyOS experience. The third group consisted of the four developers that have worked on the framework, the most junior one having six months of TinyOS experience. Each of those diverse groups provided us with valuable feedback.

8.3.1. Quantitative results. A major obstacle to adopting unit testing is the overhead that unit testing incurs on a software development process. To check if a module works as designed, unit tests for the module have to be developed, executed, and assessed. Usually these activities are repeated multiple times until all the tests pass and it has been ascertained that they sufficiently cover the module's code. While the solutions offered by our framework make test execution and assessment as inexpensive as executing a single command, test development requires effort from programmers. Since the development effort actually determines the overall cost of unit testing, we study to what extent our testing model reduces the cost of test development.

Selecting the metric that best quantifies the development overhead is not trivial, though. We have settled on a metric that is easily measurable and verifiable: the number of lines of code (LOC).[1] However, in addition, we offer a discussion on the actual development time.

As the first example, we illustrate the amount of LOC that must be written to thoroughly test a complete medium-scale subsystem. The study was performed in the first group working on an aggregation protocol for mobile networks. The two developers were writing subsequent modules of the protocol in parallel with unit tests that fully cover the control flow paths of the modules. Table V illustrates the breakdown of LOC for the complete code, excluding applications demonstrating the use of the protocol (de facto integration tests for the protocol).

Table V: Sample unit testing overhead of medium-scale protocol

SOFTWARE COMPONENT	LINES OF CODE (LOC)
all tested protocol modules (just modules w/o interfaces, etc.)	1180
all test cases – parts written manually – parts generated automatically	2065 1185 880
mock skeletons (generated automatically)	132∗
remaining test code (e.g., glue code, makefiles, scripts, etc.) – parts written manually – parts generated automatically	2863 21 2842
total test code – parts written manually – parts generated automatically	5060 1206 3854

∗ Just mock skeletons are taken into account, such as the one in Listing 10. The actual code generated from the skeletons by the extended nesC compiler is significantly larger.

The subsystem implementing the protocol consists of 14 modules that together contain 1180 LOC. The modules interact with each other and TinyOS through 31 interfaces, 8 of which are standard TinyOS interfaces for packet manipulation, sending and receiving, for timer access, for generating random numbers, and for activating and deactivating components. The unit tests for the modules comprise 14 suites (one per module) amounting to 54 test cases. The test cases contain

[1] Measured with CLOC: http://cloc.sourceforge.net/.

in total 2065 LOC, 1185 of which is test logic written by the programmers. The remaining test code comprises automatically generated skeletons of embedded mock modules, test-suite glue code, configurations, and scripts.

A major conclusion derivable from Table V is that the total amount of test code exceeds the amount of the actual protocol code by more than a factor of four. This illustrates that unit testing may indeed be expensive in terms of the test development effort. However, with the support from our framework, only 1206 of the test code lines have to be implemented manually, which brings the test development overhead to slightly more than a factor of one, or even less if we counted interface definitions as the module code. Moreover, the programmers reported (albeit without providing us with concrete numbers) that, compared to the module code, developing the test logic consumed significantly less time than what the LOC results suggest. They attributed this phenomenon to the fact that, in contrast to protocol code that must take actions in response to events, the test code typically boils down to verifying the flow and effects of those actions. Whereas the actions can be quite complex, the verification of their effects can often be done with just a few expectations and assertion macros. By and large, the framework simplifies unit test development.

The next example illustrates to what extent the automatically generated embedded mock modules reduce the test development overhead. The study was conducted by two of the people working on the framework itself, and, in contrast to the previous one, involved a standard subsystem, namely the popular Four-Bit Wireless Link Estimator (4bitLE) [Fonseca et al. 2007]. The TinyOS implementation of 4bitLE is a single 487-line module, `LinkEstimatorP`, that provides 7 and uses 6 interfaces, and encompasses link table maintenance, link quality arithmetic, estimation logic, beacon packet decoration and parsing, and counting received and missed packet acknowledgments. It quickly became apparent that unit tests fully covering such a monolithic implementation would be large and difficult to write. Consequently, the module was decomposed into six modules corresponding to the aforementioned responsibilities and an extra glue module: 584 lines in total. In parallel, unit tests for the module were developed in our framework.

With those tests in place, we asked the developers to write another set of tests that, rather than relying on the automatically-generated embedded mock modules, employ manually-written stub modules. The aim of the exercise was to compare the savings in LOC provided by embedded mock modules alone. Note that we deliberately conducted the exercise in the group of developers working on the framework. This was because the developers knew well the internals of embedded mock modules and were in position to optimize the manually-written stubs to make the comparison as fair as possible. Table VI presents the relevant LOC statistics.

Table VI: Sample LOC savings due to embedded mock modules

	LINES OF CODE (LOC)	
	manually-written stub modules	**embedded mock modules**
tested modules	584	
manually-provided stubs	1430	0
test cases	940	834
overhead factor of only stubs	**2.45**	**0**

The results suggest that embedded mock modules indeed significantly reduce the cost of developing unit tests. In particular, in the considered example, the manually written stub modules alone generate a factor of 2.45 testing overhead. Moreover, they entail an additional overhead in the test logic itself, because instead of succinct expectations the developers must use invocations of functions for configuring the stubs. In analogous studies conducted for another popular TinyOS protocol, the aforementioned Collection Tree Protocol (CTP) [Gnawali et al. 2009], we obtained similar results. In particular, in the core modules of CTP, `CtpRoutingEngineP` and `CtpForwardingEngineP`, after reaching a factor of four overhead due to just stub modules alone, our developers simply refused writing any more stubs.

The last example compares our framework with TUnit [Moss 2007]. The results were reported by the group working on a new hardware-software platform. The comparison was performed on a small standard UDP socket layer that consisted of a module implementing a socket and another module managing allocation of ports to sockets. The modules had few (up to four) dependencies, which is favorable for TUnit that does not automatically abstract out dependencies by means of embedded mock modules. Table VII presents the LOC statistics.

Table VII: Sample comparison of our framework with TUnit

SOFTWARE COMPONENT	LINES OF CODE (LOC)	
	OUR FRAMEWORK	TUNIT
all tested protocol modules	216	
mock/stub modules	12	281
- generated automatically	12	0
- written manually	0	281
test cases and remaining test code	476	296
– parts written manually	119	296
– parts generated automatically	357	0
total manually written test code	119	577
overhead factor	0.55	2.67

The example shows that, in terms of LOC, our framework outperforms TUnit by nearly a factor of five: its overhead is just a factor of 0.55, whereas the overhead of TUnit is a factor of 2.67. The manually-written stubs that TUnit requires alone contribute significantly to this overhead. Moreover, note that the tested modules were simple, meaning that they involved relatively few control flow paths and just 216 LOC. As a result, the stubs implemented for TUnit were also simple. Had there been more control paths or dependencies in the tested code, the stubs would have also required more LOC, thereby performing significantly worse than our framework (c.f. the previous example).

Another observation is that, if stub modules are ignored, our framework actually requires more test code than TUnit (476 LOC vs. 296 LOC). This is an artifact of our testing model that assumes isolation of the tested code and defines how test cases are implemented and executed. Supporting our model thus may require more code than supporting a more general model of TUnit. At the same time, however, because of being more stringent, our model allows for generating lots of the test code automatically, including test case skeletons, glue code, makefiles, and execution scripts. In contrast, generating such code for TUnit would be more complicated. In effect, the actual programming overhead is in fact lower in our model (119 LOC vs. 296 LOC). Consequently, even modules that lack any dependencies require more manually written test code in TUnit than in our framework.

All in all, although a larger-scale study would be necessary to produce more statistically accurate numbers, there is every reason to believe that embedded mock modules indeed improve unit test development in sensornets. To start with, writing stub code is no longer required, which is extremely important considering the aforementioned number of dependencies of an average module. Moreover, the new language constructs associated with embedded mock modules facilitate concisely expressing the verified properties of the tested code, which shortens test cases. Finally, the test case model enabled by the isolation due to the mock modules allows for automatically generating lots of support code and test case skeletons, which further reduces the test development overhead. What remains to be implemented by developers is thus pure test logic, which in addition can usually be expressed with succinct expectations and assertion macros. Such an extent of support for unit testing has to date been missing in sensornet programming environments.

8.3.2. Qualitative results. Equally important as the quantitative benefits of our framework are qualitative ones. We have gathered the qualitative results from interviews with the members of the three aforementioned groups evaluating the framework. We were interested in their experience adopting unit testing to sensornets and whether the framework actually helped in the process. Overall, the interviews confirmed all the aforementioned qualities of unit testing, though we were sur-

prised to learn that some aspects were more important than we had initially assumed. Let us start with case studies of software defects revealed by our framework.

Case study 1: Internal MCU flash storage. One of the tasks of the four developers working on the new hardware-software platform was to build a storage system for the internal flash of an MCU. The system uses segments of the flash that are unoccupied by the program running on the MCU for storing a circular log of variable-sized records. The segments in which the records are stored are known to the storage system, but need not be contiguous. Worse yet, they can be interleaved with the program the MCU runs, thus also the code implementing the storage system itself. Consequently, any flaw in the implementation of the system could cause an overwrite of the software image run by the MCU, which could lead to an arbitrary behavior. For this reason, it was vital for the developers to ensure that before actually writing anything to the flash, their implementation does not have any flaws, to which end they employed our framework from the start of the development process.

The framework indeed helped avoid problems. In particular, in the initial implementation, the logical format of a log record was as in Fig. 9(a). A record thus conceptually consisted of a one-byte magic number, a payload of an arbitrary length, a two-byte record sequence number, a two-byte CRC, and again the magic number. However, the magic-number byte was not allowed to appear anywhere in the record apart from the two designated places. Therefore, special encoding-decoding software components were necessary to escape any bytes equal to the magic number in the payload, sequence number, and CRC, meaning that these fields could in practice occupy more bytes than in the conceptual format (up to two times more). Without getting into details, such a format was chosen to enable recovering a sequence of undamaged records for post-deployment analyses. What is important is that the implementation was accompanied by unit tests with 100% coverage.

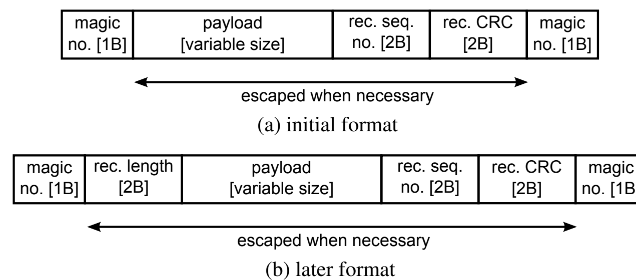

Fig. 9: Format of records in log storage based on internal MCU flash

With the initial implementation operational, it turned out that appending to the log after a node reboot would be useful functionality, for which, however, faster seeking for the end of the record sequence was necessary. To this end, the conceptual record format was extended with a two-byte record length field immediately following the leading magic number, as in Fig. 9(b).

Albeit seemingly simple, that change introduced two flaws. First, while the encoding and decoding code correctly escaped the new record length field, the new seeking code assumed that the field was two bytes. In effect, if there was a record with an escaped length, after a node reboot, log appending could start in an arbitrary place, thereby corrupting the existing data. Second, the code that checked whether a appended record did not overflow the current flash fragment also did not take into account the escaping of the record length field. In that case, the results of running such code could be even more grave, as an overflowing record could overwrite up to two bytes of the node program image. Both the flaws were detected early with the available unit tests.

Although in retrospect the flaws seem evident, they were not during the development process. Moreover, they would be unlikely to be discovered with just, for instance, integration testing, because to be escaped, a byte in the record length field had to be equal to a specific value (either the magic number or the escape symbol), and the likelihood of such an event was not high. Even if a

flaw, notably the second one, occurred in integration testing, reproducing it would be a difficult task. Diagnosing it would be equally daunting, as any modifications introduced to the program image to perform the diagnosis would likely change the flash fragments available to the storage system. In contrast, unit testing turned out to be ideal for detecting these types of implementation defects.

Case study 2: In-network aggregation protocol. Similar conclusions were reached by the group working on the aforementioned aggregation protocol. Although the details of the protocol are out of the scope of this article, at a high level, the operation of the protocol involves three phases corresponding to the following software components: an initializer, an aggregator, and an evaluator. The initializer checks if local node data satisfy a query and, if so, creates a partial aggregate for the data. The aggregator is invoked whenever a node receives a partial aggregate from another node to combine the received aggregate with the node's local aggregate. The evaluator turns the local partial aggregate into a human-friendly result of the query. In addition, there are several components that run the distributed aggregation process, prepare the messages, etc.

Unit testing with our framework identified the following defects in the components:

(1) incorrect bit shifts in a custom fixed-point-arithmetic multiplication function of the evaluator, which produced invalid query results for some classes of numbers;
(2) mixing messages from different aggregation epochs in the combiner, which produced wrong partial aggregates;
(3) aggregating inconsistent data received from another node in the combiner, which on rare occasions could disrupt partial aggregates;
(4) invoking a packet manipulation function from a wrong layer of the protocol stack, which could lead to buffer overflows at run time;
(5) an epoch timer not being started;
(6) an equality instead of inequality operator in a conditional instruction in a corner case related to neighbor selection, which could results in hard-to-detect suboptimal selections.

Had the developers skipped unit testing and immediately proceeded to integration testing, they would have been able to observe only that the produced query results are wrong or that no results are produced at all in some cases. Identifying the listed defects as the source of such behavior would be extremely laborious, and it is not clear whether all of the defects would have been identified. For example, according to the developers, an investigation of invalid query results (defect 1) would have likely started from the components responsible for communication, and would have later followed to the initializer and the combiner. Arithmetic operations in the evaluator, in contrast, would have been one of the last places to look at, especially as the defect was triggered only for some classes of numbers. Likewise, defects 3 and 6 would have been difficult to diagnose with only integration testing, as they would have manifested only occasionally. In contrast, unit testing eliminated those defects early: when some of the components were not even implemented. In effect, the resulting protocol worked out-of-the-box in the first deployment, after parameter fine-tuning.

Note, however, that some of the defects, namely 2, 3, and 4, were detected by unit testing configurations that involved several modules rather than testing individual modules. In those cases, testing configurations simply made more sense and required less effort than testing individual modules.

General observations. In summary, for all group members, the most important advantage of unit testing was the confidence they gain when their code is accompanied by unit tests. The development support offered by the framework allows for quickly assembling test cases that exercise arbitrary control flow paths in the code. The assessment support, in turn, enables effectively checking if any corner cases are not yet covered by tests. As a result, the developers could exercise every line and branch in their modules, even when other modules were not ready.

Somewhat surprising is the fact that another most frequently reported advantage of unit testing is the existence of tests themselves. Developers that joined the projects later reported that the self-descriptive, expectation-based tests helped them understand code, and subsequently modify it without breaking the entire systems. Likewise, developers maintaining mature code claimed that the

existence of tests protected against introducing defects to the code when adding new features, like in the first case study. In contrast, existing sensornet code bases contain relatively few tests that illustrate how the more involved components work internally and collaborate with other components. Had they been accompanied by unit tests, their learning curves might have been less steep.

Another major benefit of framework-assisted unit testing is the effects on the resulting software. More specifically, in large, monolithic modules, it is hard to exercise all control flow paths. In contrast, designing for unit testability, be it by means of TDD or in a more traditional development process, promotes small modules with well-defined functionality. To illustrate, recall the aforementioned aggregation protocol which involved 1180 LOC. The protocol was implemented by 14 modules. In contrast, as the example of the 4BitLE illustrates, it is not uncommon for TinyOS modules to contain as much as 500 LOC. Understanding the behavior of such large modules need not be easy. Likewise, reusing pieces of code becomes problematic with large modules. Small modules accompanied by self-descriptive unit tests alleviate these issues.

The developers also greatly appreciated the support for test execution. The virtually immediate feedback on test case failures allowed them to quickly diagnose and fix their software flaws, in most cases without manually inserting debug statements or employing external debuggers. We learned that most of the testing was done in TOSSIM, which made the execution process fast, and only when tests passed in TOSSIM were they run on actual sensor nodes. The developers claimed that the ability to quickly find and diagnose software defects allowed them to stay focused on the code and accelerated the entire development process, which is important particularly for TDD.

Although many of the above conclusions correspond to good software engineering practices, the existing code bases and past deployments evidence that we, as a community, have often failed to apply these practices in our sensornet software development processes. We strongly believe that this is a consequence of the lack of appropriate unit testing solutions, which made applying many of the practices simply uneconomic. The goal of our work is to fill this gap by bringing state-of-the-art unit testing techniques for traditional software to sensornets. The interviews suggest that the goal has been achieved to a large extent, as all the developers reported that their experience with our framework resembles what they were used to with modern frameworks for traditional software.

8.4. Limitations

Despite the aforementioned advantages, our framework has a few limitations. Whereas some of the limitations are artifacts of our solutions, others stem from the very nature of unit testing itself. Let us thus briefly review the most important ones.

To begin with, the isolation of tested code from the underlying hardware, which our framework promotes, precludes testing the low-level hardware presentation layer of device drivers. This implies that, for a fraction of code, no tests can be developed with our framework. Neither is the framework able to detect situations in which the hardware is driven into violating its specification, which may result in some unexpected behavior. We take the stand that complete device drivers should be tested in integration tests, which, as mentioned throughout the article, our framework by no means aims to replace. Nevertheless, as also mentioned previously and confirmed by the participants of our user study who were involved in developing device drivers, the framework encourages thin hardware presentation layers, without unnecessarily incorporating the logic of higher layers. As a result, most of a driver's logic resides in these higher layers and can thus be unit tested as ordinary modules. For this reason, a lot of the driver's code will typically already be correct before deployment on the actual hardware, which overall reduces the time required to make the driver stable.

Moreover, the framework could benefit from being integrated with a development environment, like Eclipse. Such integration could further simplify writing, executing, and assessing test cases, thereby making the presented solutions more accessible, especially to novice users. Considering that a particularly important drawback of unit testing itself is that it requires test code to be developed, an integrated environment could reduce the costs of unit testing further beyond what is offered by our framework. Nevertheless, we are aware that unit testing does require additional effort. Therefore,

we envision that our framework will be adopted rather for new code. Existing code, in turn, may be covered gradually with tests, as a side effect of refactoring or implementing new functionality.

All in all, despite its limitations, unit testing plays an important role in the spectrum of the software quality assurance techniques. Our framework facilitates adopting it in sensornets.

9. CONCLUSIONS

To sum up, our work demonstrates that bringing modern unit testing techniques to sensornets poses a number of challenges, but the benefits such techniques offer may be worth the effort. To begin with, mocking functionality, a fundamental modern code isolation mechanism, in sensornets necessitates language and compiler enhancements. However, it reduces the test development overhead even by several factors and makes the resulting test cases self-descriptive. Likewise, in the absence of traditional hardware-software support, intercepting and containing run-time failures to provide meaningful feedback requires dedicated sensornet solutions. However, such solutions spare the effort involved in deploying debuggers to diagnose many classes of software defects. Similarly, code coverage measurement, a popular test assessment method, is virtually impossible without dedicated sensornet solutions. However, the solutions evidence, in particular, that integration tests, even carefully designed ones, fail to cover many control flow paths, thereby substantiating the claim about a need for unit testing. Finally, the proposed solutions also facilitate effective management of the quality of sensornet software. In particular, they allow for ensuring that, even in the face of constant changes by multiple developers, the software is accompanied by thorough unit tests.

As a whole, our work finally enables modern software development methodologies, such as TDD, in sensornets. Since these methodologies are particularly popular for building reliable systems software, adopting them has the potential to greatly improve the quality of future sensornet systems, which are likely to be more complex than any deployments to date. For this reason, we plan to make the framework publicly available. It would be interesting to observe at scale how the proposed solutions affect sensornet software development in the long term, in particular, whether they can make the quality of sensornet software more predictable. Subsequent components of the framework will gradually be made available for download from the corresponding author's website.

ACKNOWLEDGMENTS

The authors would like to thank the associate editor, Tarek Abdelzaher, and the anonymous reviewers whose comments have helped to improve the quality of this article.

REFERENCES

E. Akhmetshina, Pawel Gburzynski, and Frederick S. Vizeacoumar. 2003. PicOS: A Tiny Operating System for Extremely Small Embedded Platforms. In *ESA '03: Proceedings of the International Conference on Embedded Systems and Applications*. Las Vegas, NV, USA, 116–122.

Will Archer, Philip Levis, and John Regehr. 2007. Interface Contracts for TinyOS. In *IPSN '07: Proceedings of the 6th ACM/IEEE International Conference on Information Processing in Sensor Networks*. Cambridge, MA, USA, 158–165. DOI:http://dx.doi.org/10.1145/1236360.1236382

David Astels. 2003. *Test-Driven Development: A Practical Guide*. Prentice Hall. 592 pages.

Guillermo Barrenetxea, François Ingelrest, Gunnar Schaefer, and Martin Vetterli. 2008. The Hitchhiker's Guide to Successful Wireless Sensor Network Deployments. In *SenSys '08: Proceedings of the 6th ACM International Conference on Embedded Networked Sensor Systems*. Raleigh, NC, USA, 43–56. DOI:http://dx.doi.org/10.1145/1460412.1460418

Kent Beck. 1999. Embracing change with extreme programming. *IEEE Computer* 32, 10 (October 1999), 70–77. DOI:http://dx.doi.org/10.1109/2.796139

Nicolas Burri, Roland Schuler, and Roger Wattenhofer. 2006. YETI: A TinyOS Plug-in for Eclipse. In *REALWSN '06: Proceedings of the ACM Workshop on Real-World Wireless Sensor Networks*. Uppsala, Sweden.

Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. 2007. Dependent Types for Low-Level Programming. In *ESOP '07: Proceedings of the 16th European Symposium on Programming*. Braga, Portugal, 520–535. http://dl.acm.org/citation.cfm?id=1762174.1762221

Nathan Cooprider, Will Archer, Eric Eide, David Gay, and John Regehr. 2007. Efficient Memory Safety for TinyOS. In *SenSys '07: Proceedings of the 5th ACM International Conference on Embedded Networked Sensor Systems*. Sydney, Australia, 205–218. DOI:http://dx.doi.org/10.1145/1322263.1322283

Mark Doliner. 2006. Cobertura Homepage. online: http://cobertura.sourceforge.net/. (2006).

Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. 2004. Contiki – A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *LCN '04: Proceedings of the 29th Annual IEEE Conference on Local Computer Networks*. Tampa, FL, USA, 455–462. DOI:http://dx.doi.org/10.1109/LCN.2004.38

Emre Ertin, Anish Arora, Rajiv Ramnath, Vinayak Naik, Sandip Bapat, Vinod Kulathumani, Mukundan Sridharan, Hongwei Zhang, Hui Cao, and Mikhail Nesterenko. 2006. Kansei: A Testbed for Sensing at Scale. In *IPSN '06: Proceedings of the 5th International Conference on Information Processing in Sensor Networks*. Nashville, TN, USA, 399–406. DOI:http://dx.doi.org/10.1145/1127777.1127838

Szczepan Faber. 2008. Mockito homepage. online: http://code.google.com/p/mockito/. (2008).

Rodrigo Fonseca, Omprakash Gnawali, Kyle Jamieson, and Philip Levis. 2007. Four-Bit Wireless Link Estimation. In *HotNets-VI: Proceedings of the 6th ACM Workshop on Hot Topics in Networks*. Atlanta, GA, USA. http://conferences.sigcomm.org/hotnets/2007/papers/hotnets6-final131.pdf

Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. 2004. Mock Roles, Not Objects. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. Vancouver, Canada, 236–246. DOI:http://dx.doi.org/10.1145/1028664.1028765

David Gay, Philip Levis, and David Culler. 2007. Software design patterns for TinyOS. *ACM Transactions on Embedded Computing Systems* 6, 4 (September 2007), 22:1–22:39. DOI:http://dx.doi.org/10.1145/1274858.1274860

David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. 2003. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. San Diego, CA, USA, 1–11. DOI:http://dx.doi.org/10.1145/781131.781133

Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. 2009. Collection Tree Protocol. In *SenSys '09: Proceedings of the 7th ACM International Conference on Embedded Networked Sensor Systems*. Berkeley, CA, USA, 1–14. DOI:http://dx.doi.org/10.1145/1644038.1644040

Google. 2008. Googlemock Homepage. online: http://code.google.com/p/googlemock/. (2008).

Brian J. Gough. 2004. *An Introduction to GCC: For the GNU compilers gcc and g++*. Network Theory Ltd., Chapter 10.3. http://www.amazon.com/exec/obidos/ASIN/0954161793/networktheory-20

Michal Gregorczyk, Tomasz Pazurkiewicz, and Konrad Iwanicki. 2014. On Decentralized In-Network Aggregation in Real-World Scenarios with Crowd Mobility. In *DCOSS 2014: Proceedings of the 10th IEEE International Conference on Distributed Computing in Sensor Systems*. IEEE, Marina del Rey, CA, USA, 83–91. DOI:http://dx.doi.org/10.1109/DCOSS.2014.8

Konrad Iwanicki, Albana Gaba, and Maarten van Steen. 2008. *KonTest: A Wireless Sensor Network Testbed at Vrije Universiteit Amsterdam*. Technical Report IR-CS-045. Vrije Universiteit Amsterdam, Amsterdam, the Netherlands. URL: http://www.mimuw.edu.pl/%7Eiwanicki/publications/.

Konrad Iwanicki and Maarten van Steen. 2007. Towards a Versatile Problem Diagnosis Infrastructure for Large Wireless Sensor Networks. In *OTM 2007 Workshops: Proceedings of the 2nd Workshop on Pervasive Systems (PerSys)*. Springer-Verlag LNCS 4806, Vilamoura, Portugal, 845–855. DOI:http://dx.doi.org/10.1007/978-3-540-76890-6_10

Nupur Kothari, Todd Millstein, and Ramesh Govindan. 2008. Deriving State Machines from TinyOS Programs Using Symbolic Execution. In *IPSN '08: Proceedings of the 7th ACM/IEEE International Conference on Information Processing in Sensor Networks*. St. Louis, MO, USA, 271–282. DOI:http://dx.doi.org/10.1109/IPSN.2008.62

Lakshman Krishnamurthy, Robert Adler, Phil Buonadonna, Jasmeet Chhabra, Mick Flanigan, Nandakishore Kushalnagar, Lama Nachman, and Mark Yarvis. 2005. Design and Deployment of Industrial Sensor Networks: Experiences from a Semiconductor Plant and the North Sea. In *SenSys '05: Proceedings of the 3rd ACM International Conference on Embedded Networked Sensor Systems*. San Diego, CA, USA, 64–75. DOI:http://dx.doi.org/10.1145/1098918.1098926

Koen Langendoen, Aline Baggio, and Otto Visser. 2006. Murphy Loves Potatoes: Experiences from a Pilot Sensor Network Deployment in Precision Agriculture. In *IPDPS '06: Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*. IEEE, Rhodes Island, Greece, 155. DOI:http://dx.doi.org/10.1109/IPDPS.2006.1639412

Philip Levis, David Gay, Vlado Handziski, Jan-Heinrich Hauer, Ben Greenstein, Martin Turon, Jonathan Hui, Kevin Klues, Cory Sharp, Robert Szewczyk, Joseph Polastre, Philip Buonadonna, Lama Nachman, Gilman Tolle, David Culler, and Adam Wolisz. 2005. *T2: A Second Generation OS For Embedded Sensor Networks*. Technical Report TKN-05-007. Technische Universität, Berlin, Germany. http://sing.stanford.edu/pubs/tinyos2-tr.pdf

Philip Levis, Nelson Lee, Matt Welsh, and David Culler. 2003. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *SenSys '03: Proceedings of the 1st ACM International Conference on Embedded Networked Sensor Systems*. Los Angeles, CA, USA, 126–137. DOI:http://dx.doi.org/10.1145/958491.958506

Peng Li and John Regehr. 2010. T-Check: Bug Finding for Sensor Networks. In *IPSN '10: Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. Stockholm, Sweden, 174–185. DOI:http://dx.doi.org/10.1145/1791212.1791234

Mateusz Michalowski, Przemyslaw Horban, Karol Strzelecki, Jacek Migdal, Maciej Klimek, Piotr Glazar, and Konrad Iwanicki. 2012. *A Sensornet Testbed at the University of Warsaw*. Technical Report TR-DS-01/12. University of Warsaw, Warsaw, Poland. URL: http://www.mimuw.edu.pl/%7Eiwanicki/publications/.

David Moss. 2007. TUnit Documentation. online: http://docs.tinyos.net/tinywiki/index.php/TUnit. (2007). http://docs.tinyos.net/tinywiki/index.php/TUnit

George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*. Springer-Verlag, Grenoble, France, 213–228. DOI:http://dx.doi.org/10.1007/3-540-45937-5_16

Michael Okola and Kamin Whitehouse. 2010. Unit Testing for Wireless Sensor Networks. In *SESENA '10: Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*. Cape Town, South Africa, 38–43. DOI:http://dx.doi.org/10.1145/1809111.1809123

Fredrik Österlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. 2006. Cross-Level Sensor Network Simulation with COOJA. In *SenseApp '06: Proceedings of the 1st IEEE International Workshop on Practical Issues in Building Sensor Network Applications*. Tampa, FL, USA, 641–648. DOI:http://dx.doi.org/10.1109/LCN.2006.322172

Tomasz Pazurkiewicz, Michal Gregorczyk, and Konrad Iwanicki. 2014. NarrowCast: A New Link-layer Primitive for Gossip-based Sensornet Protocols. In *EWSN 2014: Proceedings of the 11th European Conference on Wireless Sensor Networks*. Springer-Verlag LNCS 8354, Oxford, UK, 1–16. DOI:http://dx.doi.org/10.1007/978-3-319-04651-8_1

Nithya Ramanathan, Eddie Kohler, and Deborah Estrin. 2005. Towards a debugging system for sensor networks. *International Journal of Network Management* 15, 4 (July 2005), 223–234. DOI:http://dx.doi.org/10.1002/nem.570

Kay Romer and Junyan Ma. 2009. PDA: Passive Distributed Assertions for Sensor Networks. In *IPSN '09: Proceedings of the 8th ACM/IEEE International Conference on Information Processing in Sensor Networks*. San Francisco, CA, USA, 337–348. http://dl.acm.org/citation.cfm?id=1602165.1602196

Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. 2010. KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks before Deployment. In *IPSN '10: Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. Stockholm, Sweden, 186–196. DOI:http://dx.doi.org/10.1145/1791212.1791235

Ken Schwaber and Mike Beedle. 2001. *Agile Software Development with Scrum* (first ed.). Prentice Hall.

Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. 2005. Avrora: Scalable Sensor Network Simulation with Precise Timing. In *IPSN '05: Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*. Los Angeles, CA, USA, 477–482. DOI:http://dx.doi.org/10.1109/IPSN.2005.1440978

Gilman Tolle and David Culler. 2005. Design of an Application-Cooperative Management System for Wireless Sensor Networks. In *EWSN '05: Proceedings of the 2nd European Workshop on Wireless Sensor Networks*. IEEE, Istanbul, Turkey, 121–132. DOI:http://dx.doi.org/10.1109/EWSN.2005.1462004

Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. 2006. Fidelity and Yield in a Volcano Monitoring Sensor Network. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*. Seattle, WA, USA, 381–396. http://www.usenix.org/events/osdi06/tech/werner-allen.html

Geoffrey Werner-Allen, Patrick Swieskowski, and Matt Welsh. 2005. MoteLab: A Wireless Sensor Network Testbed. In *IPSN '05: Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*. Los Angeles, CA, USA. http://dl.acm.org/citation.cfm?id=1147685.1147769

Kamin Whitehouse, Gilman Tolle, Jay Taneja, Cory Sharp, Sukun Kim, Jaein Jeong, Jonathan Hui, Prabal Dutta, and David Culler. 2006. Marionette: Using RPC for Interactive Development and Debugging of Wireless Embedded Networks. In *IPSN '06: Proceedings of the 5th International Conference on Information Processing in Sensor Networks*. Nashville, TN, USA, 416–423. DOI:http://dx.doi.org/10.1145/1127777.1127840

Jing Yang, Mary Lou Soffa, Leo Selavo, and Kamin Whitehouse. 2007. Clairvoyant: A Comprehensive Source-Level Debugger for Wireless Sensor Networks. In *SenSys '07: Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*. Sydney, Australia, 189–203. DOI:http://dx.doi.org/10.1145/1322263.1322282

Received April 2013; revised February 2014; accepted March 2014