

# On Mining Sensor Network Software Repositories

Andreas Loukas  
Embedded Software group  
Delft University of Technology  
The Netherlands  
a.loukas@tudelft.nl

Matthias Woehrle  
Embedded Software group  
Delft University of Technology  
The Netherlands  
m.woehrle@tudelft.nl

Koen Langendoen  
Embedded Software group  
Delft University of Technology  
The Netherlands  
k.langendoen@tudelft.nl

## ABSTRACT

Wireless Sensor Network (WSN) software is typically developed in one of the two prominent WSN operating systems: TinyOS or Contiki. Both of these operating systems are open-source projects and basically frameworks for WSN developers. In this paper, we study the software repositories of these two projects. Software repositories provide a wealth of information on software projects and their development. Based on the mined information, we explore the TinyOS and Contiki commit history and compare them to an open-source embedded operating system, Ethernut. As a second step, we explore WSN-specific artifacts and mine TinyOS software for cross-cutting concerns. Most of the relations we find are not cross-cutting. Nevertheless, we do find cross-cutting concerns that are resource-related.

## Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement

## General Terms

Measurement, Human Factors

## Keywords

Wireless Sensor Networks, Mining Software Repositories, Cross-Cutting Concerns

## 1. INTRODUCTION

The history stored in software repositories provides a rich source of information that can be mined to gain insight into software projects. For Wireless Sensor Networks (WSNs), there are two prominent software projects: TinyOS<sup>1</sup> and Contiki<sup>2</sup>. Both of these WSN operating systems have been in active development for multiple years and, hence, provide

<sup>1</sup>[www.tinyos.net](http://www.tinyos.net)

<sup>2</sup><http://www.sics.se/contiki/>

Permission to make digital or hard copies of all or part 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 bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA  
Copyright 2011 ACM 978-1-4503-0583-9/11/05 ...\$10.00.

historical data on the software and its development process. In this paper, we mine the software repositories of these two projects. We discuss our methodology based on file-level mining and present initial results on mining the commit history w.r.t. the software and its development process. Additionally, we compare these projects to an embedded project: Ethernut<sup>3</sup>.

As a second step, we look at a particular artifact found in software: *cross-cutting concerns*, i.e., pieces of functionality that cannot be cleanly separated. Walton et al. [19] discuss that the constraints of sensor nodes, such as energy, processing power and memory, result in cross-cutting concerns. We mine the software repository of TinyOS to identify cross-cutting concerns and determine their cause.

The contributions of this paper can be summarized as:

- We mine the two most prominent sensor network software repositories of Contiki and TinyOS.
- We compare the differences to a standard embedded operating systems, Ethernut.
- We identify and discuss cross-cutting concerns in sensor network software [19].

In the following section we present the three operating systems and introduce mining of software repositories. In Sec. 3, we perform a comparative analysis of the project repositories. We study and discuss cross-cutting concerns in TinyOS in Sec. 4. Sec. 5 concludes our paper.

## 2. BACKGROUND AND RELATED WORK

In this section, we describe the three software projects used in our study and detail on related work in mining software repositories.

### 2.1 Open-Source, Embedded Operating Systems

First we describe the two WSN operating systems. Note that these operating systems typically include various functionality: the kernel, device drivers for microcontrollers, radios and sensors, different protocols primarily focused on the MAC and the network layer, developer support such as shells, debugging primitives, programming and file system support. Importantly, they also feature example applications for standard WSN functionality, in particular data collection. Recent developments in the 6lowpan standard, have also resulted in an increased interest in corresponding protocol stacks in both operating systems. Note that in the following discussion we are not focussing on the model of

<sup>3</sup><http://ethernut.de/>

Repository	TinyOS	Contiki	Ethernut
Class	WSN	WSN	Embedded
Period	10/04-01/11	01/06-12/10	08/01-03/09
Files	3119	1240	1000
LOC <sup>†</sup>	69	90	148
Developers	39	25	15
Transactions	2262	1989	678
Commit/File*	2	2	2
File/Commit*	2	2	2

**Table 1: Overview of the characteristics of the three software projects.** \* The table displays the median for each entry. <sup>†</sup> LOC denotes the median lines of code per file.

execution of the operating systems, but rather the structure of their software implementation.

**TinyOS** [12] is an operating system for wireless sensor nodes. TinyOS is implemented in nesC [6] and based on the concept of modular components and interfaces. This is very similar to hardware description languages where components are (statically) wired together to form a high-level functionality. A component is implemented in a separate file (`ComponentP.nc`) and its wiring in a corresponding *configuration* (`ComponentC.nc`). Components use defined interfaces, e. g., standard interfaces defined in the TinyOS library. TinyOS is the oldest WSN operating system. However, we focus on TinyOS 2 [14], a major overhaul of TinyOS that had its initial commit in October 2004. TinyOS development recently changed hosting services; we look at the current repository<sup>4</sup> that includes the imported history from the original code. However, this switching of hosting services requires special considerations, e. g., the relocation of the repository induces artificial move commits and some developer names differ across the hosting platforms. To this end, we manually track the mapping of developer IDs across the different systems and filtered out automated commits.

**Contiki** [3] is a lightweight WSN operating system with specific consideration for dynamic loading and replacement of services at runtime. Contiki is written in (plain) C.<sup>5</sup> The individual services of the OS are also modular. As an example, a given MAC protocol is implemented in a separate file with a representative name.

**Ethernut** [5] in contrast is an embedded OS with a focus on ethernet applications on embedded devices. For this reason we choose it over Nut/OS, the bare OS used in Ethernut, since Ethernut additionally features application aspects. In this respect, Ethernut is similar to the previously described WSN projects. It features the kernel of a real-time OS, as well as device drivers and a TCP/IP stack. The OS is written in C<sup>6</sup>, in a modular fashion, similar to Contiki.

In the following, we mine the source code repositories of the three projects. The individual characteristics of the repositories are presented in Table 1, showing that all projects have a history of at least 5 years, a code base of 1000 or more files and similar commit characteristics.

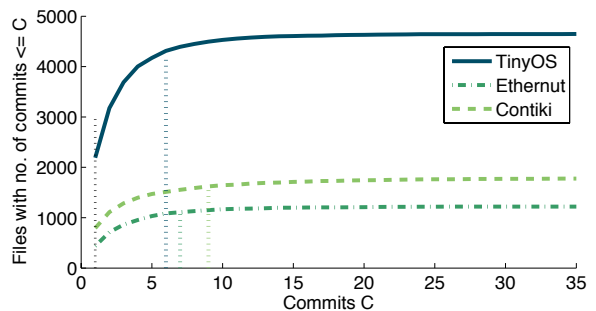
## 2.2 Mining Software Repositories

Mining software repositories (MSR) is an established field in software engineering with various interesting applications

<sup>4</sup><http://code.google.com/p/tinyos-main/>. We do not include contributed code nor TinyOS 1.

<sup>5</sup>Contiki source code is available at <http://sourceforge.net/projects/contiki/>.

<sup>6</sup>Ethernut source code is available at <http://sourceforge.net/projects/ethernut/>.



**Figure 1: Cumulative distribution of files with less than  $C$  commits. We also depict the 90% threshold, i. e., the number of commits that 90% of the files have maximally seen.**

such as guiding software changes [20], studying open-source development [2] and measuring developer contribution [9]. A comprehensive overview is out of scope of this work and the reader is referred to the annual MSR workshop series at ICSE and a special issue in the IEEE Transactions on Software Engineering [10]. Note that mining of repositories only makes sense if sufficient data is available. Since both, TinyOS and Contiki, are out of their infancy, it is time to shed light on their development, and on the intricacies and idiosyncrasies of WSN software development. As such, this paper presents an initial study on the repositories of the two main WSN operating systems.

In the following we particularly look at the commit history, where we are interested in:

- **Commit:** the change of a single file in the repository.
- **Transaction:** the change of a group of files in the repository. Each transaction contains a number of commits, an author, a log message, and a date.

Note that we mine CVS repositories with the obvious issues, e. g., file naming, commit transaction, described in [7, 17]. Since CVS does not keep track of transactions, we run a commit algorithm on the CVS history as described in [7]. Moreover, as CVS does not allow tracking of file renames, we need to devise a (simple) heuristic to catch the renames hidden in the logs. For the sake of brevity, we omit the presentation of the heuristic, yet show some results of the rename analysis in Sec. 3.4. We do not currently use branching or tagging information, except when explicitly stated, e. g., in the analysis of large commits (cf. Sec. 3.5).

## 3. ANALYZING THE REPOSITORIES

This section presents results on mining the repositories. In particular, we study the distribution of commits w. r. t. files and developers, and identify the most committed files.

### 3.1 Development Process

First we study the development process: how are commits distributed over the files of the project. Figure 1 shows the number of files that have less than  $C$  commits for TinyOS, Contiki and Ethernut, with dashed lines indicating the 90% thresholds for each project. We see three different properties: (i) The number of files considerably differs across the projects. (ii) All curves follow a similar trend. (iii) Different files are committed with significantly different frequencies. In particular for TinyOS, 90% of the files only see 6 commits, which is substantially lower than for Contiki with 9

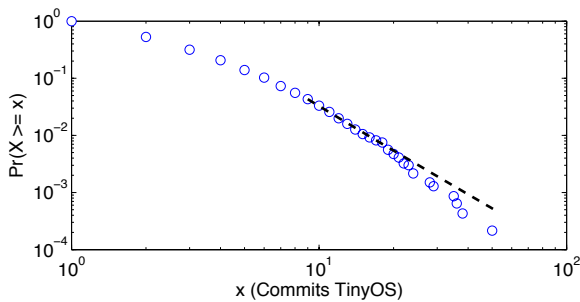


Figure 2: Maximum likelihood power-law fit (dashed) for the empirical distribution of TinyOS commits (circles) based on [1].

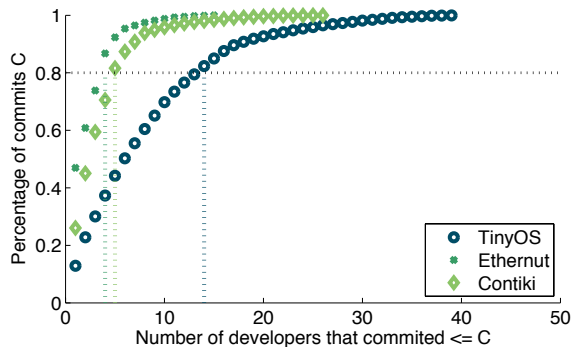


Figure 3: Empirical CDF of commits per developers. We also depict the number of developers that are responsible for 80% of the code changes.

commits. However, as we deal with the second version of TinyOS, a greater maturity is to be expected.

In order to study (ii) closer, we considered whether commits follow a power law distribution [11]: To this end, we use the methods proposed by Clauset et al. [1] to determine a maximum-likelihood fit for a power law given the empirical cumulative distribution function (CDF) of the commits. Figure 2 shows the example for TinyOS; the other two distributions display a similar trend. We see that committing does follow a power law distribution. In this respect, the repositories mined in this paper are similar to other, non-embedded open-source projects, whose commits also follow a power law distribution [11].

### 3.2 Development Community

As a second step, we want to study if the development community differs across the three projects. In particular, we want to study the hypothesis (*H1.*) described by Dinh-Trong et al. [2]: “Open source developments will have a core of developers who control the code base, and will create approximately 80 percent or more of the new functionality. If this core group uses only informal ad-hoc means of coordinating their work, the group will be no larger than 10 to 15 people.” Figure 3 depicts the number of developers that are responsible for 80% of the code changes. For TinyOS this number is the largest, with 14 for TinyOS, 6 for Contiki and 3 for Ethernut. These results confirm (*H1.*). The larger size of the TinyOS developer community is reflected in the adoption of community processes, such as the TinyOS Enhancement Protocol (TEPs) process, and in the creation of working groups that focus on specific functionality (e. g., `net`).

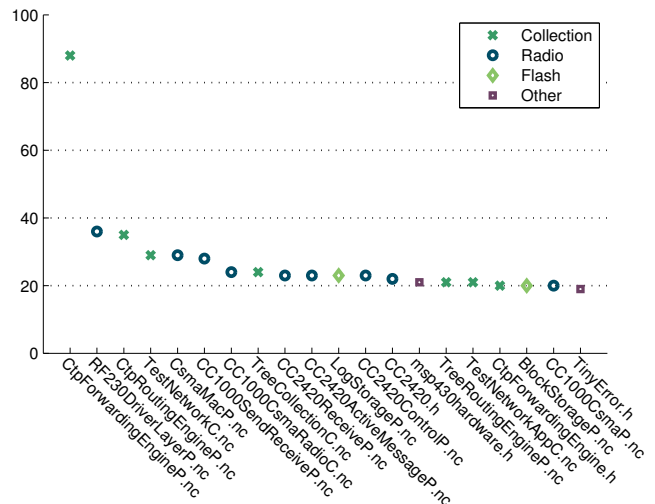


Figure 4: 20 most committed files in TinyOS and the corresponding number of commits.

### 3.3 Commits in Details

Finally, we study which files are committed the most often for the WSN projects. Figures 4 show the 20 most committed files for TinyOS<sup>7</sup>. We group them into categories and comment on their high commit number.

**TinyOS:** We group the most committed TinyOS files into 3 classes:

- (1) **CC2420 data collection stack**, i. e., files are part of the standard collection application with CTP and CC2420 radio abstractions (including tests). (`CC2420ReceiveP.nc`, `CC2420ControlP.nc`, `CC2420ActiveMessageP.nc`, `CtpForwardingEngineP.nc`, `CtpForwardingEngine.h`, `CtpRoutingEngineP.nc`, `CC2420.h`, `TestNetworkC.nc`, `TestNetworkAppC.nc`, `TreeRoutingEngineP.nc`, `TreeCollectionC.nc`)
- (2) **Support for other radios**, i. e., for RF230 (`RF230DriverLayerP.nc`), CC1000 (`CC1000CsmRadioC.nc`, `CC1000SendReceiveP.nc`, `CC1000CsmP.nc`), and TDA5250 (`CsmaMacP.nc`).
- (3) **Flash support** (`LogStorageP.nc`, `BlockStorageP.nc`)

Also often committed are the definitions for the msp430 platform (`msp430hardware.h`) and for the TinyOS error types (`TinyError.h`).

We can see a particular focus on the data collection stack, as it is the standard stack used in academia, has many users, and is therefore well tested. Additionally, as discussed in [8], CTP has been thoroughly evaluated on various platforms, testbeds and underlying MAC protocols.

**Contiki:** We group the most committed Contiki files into 5 classes:

- (1) **Data collection:** In particular for the Tmote sky and ESB. (`contiki-sky-main.c`, `contiki-msb430-main.c`, `xmac.c`, `collect.c`, `cc2420.c`)<sup>8</sup>
- (2) **IP/6lowpan** (Implementation of TCP/IP and 6lowpan in Contiki [4]) (`sicslowpan.c`, `uip.h`, `rpl-dag.c`, `rpl-icmp6.c`, `tcpip.c`, `uip.c`)
- (3) **MAC protocols** (`contikimac.c`, `simple-cc2420.c`, `lpp.c`, `xmac.c`)

<sup>7</sup>We omit Contiki and Ethernut plots for space constraints.

<sup>8</sup>We included `xmac.c` and `collect.c` twice, since they fit in both, the data collection and the MAC protocol class.

	Directory		File Name	File Ending	
Type	Add	Change	Case	Add	Change
Number	52	8	26	13	20

**Table 2: Reasons for 124 detected renames in TinyOS.**

Class	Reason	TinyOS	Contiki
Implementation	Add function	4	1
	Add platform	1	1
Corrective	Functional change	2	1
Perfective	Structural change	1	2
	Formatting	2	0
Non functional	Merge/Swap	5	0
	Change license	1	0
N/A	Various	2	0

**Table 3: Categorization of large commits for TinyOS and Contiki. (Various refers to two commits modifying a large number of files without any obvious correlation.)**

(4) **Network protocols** (`collect.c`, `uaodv.c`, `rime.c`, `rime.h`)

(5) **Programming support** (`cfs-coffee.c`, `shell-rime.c`)

Apart from these files, we also see "contiki-main.c", the main file for the c64 platform. There is generally a focus on particular platforms: the Tmote sky and the ESB platform. In comparison to TinyOS, there is a greater diversity in MAC protocols. Probably biased by its conception, Contiki focuses heavily on the 6lowpan IP stack.

An interesting observation is that both sensor network operating systems show increased commit activity in radio device drivers. This indicates that the radio drivers are considerably difficult to implement.

**Comparison to Ethernut:** In comparison to TinyOS and Contiki, the most committed files in Ethernut concern the operating system's internals, such as timer, thread, and heap implementations. Quite similar to Contiki, a large number of commits are related to TCP/IP (the Nut/Net protocol suite).

### 3.4 Detected Renames

In the following we present the results of our rename heuristic based on the TinyOS repository. We detected 119 renames with most renames featuring a Levenshtein difference  $\leq 1$  (70%). An analysis of the renames is presented in Table 2: Most changes are concerned with adding a proper ending or correcting the case of file names.

### 3.5 Large Commits

In our mining efforts, we extracted large commits with more than 100 committed files (excluding initial commits and moving repositories). We perform a classification inspired by the Extended Swanson Categories of Changes proposed in [13]. The results in Table 3 show that for TinyOS, most large commits are for adding functionality, e.g., Deluge, or merging/swapping branches. In contrast to TinyOS, Contiki has not seen many large commits.

### 3.6 Bug Tracking

In all projects bug tracking software is available, but only used little. For reference, we show the bug-tracker of Open-Embedded (OE)<sup>9</sup> a linux distribution for (larger) embedded

<sup>9</sup><http://www.openembedded.org/>

devices, which shows considerably more activity. For the WSN operating systems, bugs are often reported via mailing lists and directly fixed. Hence, establishing one-to-one correspondence between bugs and bug fixes is not directly possible. Hence, analyses determining software quality, fix latency or fault predication are not directly possible.

	TinyOS	Contiki	Ethernut	OE
Issues	280 *	18	120	2338
Since	06/2001	10/2010	05/2003	05/2005

**Table 4: Bug tracking in the projects. (\*There are two separate bug trackers with 276 and 4 bugs respectively.)**

## 4. CROSS-CUTTING CONCERNS

Walton et al. [19] discuss that the various constraints imposed by limited resources and real-time behavior in WSNs introduce *cross-cutting concerns (CCC)*. These CCCs result in relations between code modules that do not allow for clear separation (of concerns). Walton et al. propose an aspect extension to nesC to allow programmers to modularize their software. We investigate their hypothesis in the following and mine TinyOS for CCCs. Note that there are various approaches to search for CCCs [16], e.g., fan-in analysis [15]. We focus on file-level mining. Mining allows us to find relations that may not exist in the code. Nevertheless, TinyOS' naming dependencies and structure bear potential for source code analysis to improve the fidelity of our search. Note that we do not focus here on the fidelity of a specific mining approach, but investigate the existence of CCCs in the specific case of TinyOS code.

Our search for CCCs is based on the assumption that files that are frequently committed together may exhibit a (complex) semantic relation and thus constitute a CCC. Obviously, mining for CCCs may be improved by mining based on the code. However, since the TinyOS architecture is built of modular components described in individual files, our file-level mining should provide good insights into CCCs.

### 4.1 Challenges in Mining WSN Software

Embedded software architectures usually try to balance between code portability, i.e., being generic enough in order to run on different hardware, and performance, i.e., performing optimizations to reduce resource consumption. This is a trade-off that often results in strong dependencies between hardware-dependent components. In the context of CCC mining, these hardware dependencies result in relations between software components. However, many of these relations are (purely) syntactic or based on wiring respectively, rather than based on a semantic relation. There are further examples for non-semantic relations in TinyOS: There is a tight relation between a module and its configuration; the two files are implemented and updated together. Moreover, the TinyOS framework allows for writing software for different mote platforms; the portability of code necessitates a common software infrastructure between different platforms and their chips respectively. As such, there are also relations between the software for the differing hardware. Since all of these "obvious" relations are not of primary interest, we can filter them out and, hence, limit the search space and reduce the analysis complexity significantly.

## 4.2 Mining Flow

Before describing the mining flow, some definitions are needed:

- **Fileset ( $fs$ ):** a set of files that have been committed together.
- **Support:** the number of transactions that include a given fileset normalized by the total number of transactions  $T$ , denoted as  $s(fs)$ .
- **Frequent fileset (FF):** a fileset for which  $s(fs) > \frac{2}{T}$ .
- **Closed frequent fileset (CFF):** a frequent fileset is closed if it is not a subset of any other fileset with the same support.
- **Concern:** a group of frequent filesets that feature a (semantic) relation, i. e., a potential CCC.

There are two fundamental issues that are addressed in our mining flow: (i) We mine sets of files and all of their subsets, which incurs a large computational complexity. (ii) Due to the structure of our software, a lot of FFs are not real concerns. In our mining algorithm, we consider structural properties of the software to reduce the number of FFs that are not CCCs. Below we briefly describe the steps of CCC mining:

### (1) Building a transaction database.

We extract and preprocess all repository transactions. The preprocessing includes rename detection and transaction filtering (e. g., the moving commits from changing hosting services). Since many authors split their changes to consecutive transactions, we cluster transactions if consecutive transactions of an author are within a time window of 10 minutes.

### (2) Computing CFFs using the lcm [18] algorithm.

CFFs constitute a small and hence more manageable fraction of all FFs without losing any information in the process.

### (3) Filtering out irrelevant commits. (Optional)

This step allows a user to focus on a subset of files, for example platform-specific files and application-specific interfaces.

### (4) Computing the most probable FFs. (Optional)

In addition to looking at CFFs, we can also explore their subsets. However, in order to avoid an exhaustive search, we use the notion of mutual information:

$$mutual\_information(fs) = \log_2\left(\frac{s(fs)}{\prod_{i=1}^n s(file_i)}\right),$$

where  $fs$  is a fileset  $fs = [file_1, file_2, \dots, file_n]$ .

In a nutshell, the mutual information of fileset  $fs$  depends on its support, i. e., how often all files have been committed together, and on the support of its constituent files, i. e., how often each file has been committed separately. In our experiments, we use a greedy approach to determine subsets with maximal mutual information of a given cardinality.

### (5) Filtering out obvious dependencies. (Optional)

As previously discussed, a considerable fraction of the mined FFs may describe only obvious relations. In particular, in this step we exploit that in TinyOS related code is grouped into separate folders. To this end, we determine the *diversity* of each FF that is determined by the number of distinct paths of its constituent files. FFs with diversity of one are disregarded. Note that further filtering, e. g., for common interfaces, may be performed.

### (6) Grouped FFs (GFFs).

We group FFs based on their corresponding transactions. The grouping process is equivalent to finding the connected components of a graph, where nodes are FFs and edges connect the FFs that appear in common transactions. This

step facilitates FF analysis and concerns identification by partitioning unrelated FFs to different groups. The resulting GFFs however, often carry more than one concern. The interpretation of these compound concerns requires domain-specific knowledge and thus cannot be fully automated. As described in [16], identifying even simple CCCs is difficult.

## 4.3 Results

We mined the TinyOS 2 repository using the mining flow mentioned above. A total of 975 FFs and 41 GFFs were discovered, most of which describe hardware dependencies and are due to corresponding duplication of code. As described in Sec. 4.1, changes in hardware-dependent software may have to be performed for various mote platforms (e. g., changing the radio interface for different chips such as the CC2420, CC1000 and RF230). This results in relations between software across platforms that are not of primary interest for our analysis of CCCs. Hence, we remove the dominant effect of these hardware dependencies by focusing only on a single platform, TelosB. This allows for a drastic reduction in filesets, resulting in 192 FFs and 16 GFFs.

The analysis of the GFFs exposed 25 concerns, each of which reveals a dependency among a fileset. For convenience, we classify the resulting concerns in classes and specify class relation type:

- **Resource** class. Similarly to TinyOS's definition, the term resource is used in an abstract way, e. g., for power and memory management - *semantic relation*.
- **Software** class, in particular due to concerns in protocols - *semantic relation*.
- **Hardware** class, sub-classed according to the specific hardware type - *hardware relation*.
- **Programming-paradigm** class, such as glue-code or interfaces - *syntactic relation*.

These classes are not disjoint: each concern is a member of one or more classes. The classes are ordered according to relation type and all results are summarized in Table 5. On the left of the table, we place semantic relations, i. e., resource and software. These are more interesting as they constitute potential CCCs.

The resource class, in particular, exhibits the strongest cross-cutting characteristics. This confirms the hypothesis of Walton et al. [19]. We found three types of resource CCCs: memory, arbitration, and power management related.

An example of a memory-related resource CCC is message handling. In order to avoid copying buffers between the different layers of the communication stack, TinyOS uses a message buffer abstraction, `message_t`, and allows access to its fields only through a few specific interfaces (e. g., `Packet`, `AMPacket`). One of the mined concerns describes this dependency between the interfaces, the radio and serial stacks, and various other network protocols (i. e., CTP, LE, LQI). This is an obvious result of cross-layer optimization.

The largest of all mined concerns featured resource arbitration related dependencies. While examining the concern related FFs, we noted that on most occasions, arbitration refactoring caused a wide-spread propagation of changes. In essence, most of the chip-specific components, such as the MSP430's ADC and UART, the CC2420 radio stack, the STM25P flash chip, and the SHT11 humidity sensor, are tightly dependent on TinyOS's resource sharing mechanisms. This results in intertwined code.

Similar to resource arbitration, we observed a very high propagation of changes resulting from structural refactorings

resources			software	hardware				programming				
power	arbiter	memory	protocol	radio	bus	flash	other	interface	glue	system	types	naming
1	1	4	11	8	6	2	6	17	8	7	4	2

**Table 5: Overview of the number of relations found in 25 concerns for the TelosB platform. The relations are divided into separate classes and sub-classes.**

and fixes in power management. This might be because in TinyOS, power management is built on top of the resource arbitration mechanism.

The right part of Table 5 presents obvious relations in concerns due to hardware and programming-paradigm relations. The most prevalent of all programming paradigm subclasses demands that an interface is part of a concern’s FFs. These relations often occur due to a widely used interface; a change of such an interface propagates to a large part of the system. Moreover, a lot of concerns contain platform-to-chip wiring, also known as glue-code. Glue-code bundles chips into a platform.

Overall, the concerns that we did find were mostly artifacts of nesC’s and TinyOS’s programming conventions. Our mining efforts suggest that TinyOS does not implement all of its concerns in a modular way. However, this is certainly not due to negligence. It is a deliberate decision to optimize the system, yet results in code being less modular and layers that are strongly connected.

## 5. CONCLUSIONS

This paper presents mining of wireless sensor network software repositories. We performed a comparative study of the commit histories of the two major WSN software systems, TinyOS 2 and Contiki, and an embedded operating system, Ethernut. Our analysis shows similarities to other open-source software, similarities and differences across projects and their communities. Moreover, we study cross-cutting concerns in sensor network software by file-level mining of TinyOS. We identify that there are an enormous number of frequent filesets, i. e., files that are commonly committed together. Most of these frequent filesets are due to structural properties of TinyOS programming such as hardware dependencies, rather than cross-cutting concerns resulting from resource constraints. In future work, we plan to refine our method to easily filter out these obvious relations in WSN software. Nevertheless, we can identify several CCCs in TinyOS by studying the relations focussing on TelosB platform. CCCs typically originate from cross-layer optimizations, e. g., packet buffer passing, and dependencies between (tightly) coupled protocol stack components, e. g., between link estimation and routing control.

**Acknowledgements:** This research is supported by the Dutch Technology Foundation STW and the Technology Programme of the Ministry of Economic Affairs, Agriculture and Innovation.

## 6. REFERENCES

- [1] A. Clauset et al. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009.
- [2] T. T. Dinh-Trong et al. The freesbd project: A replication case study of open source development. *IEEE Trans. Softw. Eng.*, 31:481–494, June 2005.
- [3] A. Dunkels et al. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proc. 29th Annual IEEE Int’l Conf. on Local Computer Networks*, pages 455–462, 2004.
- [4] M. Durvy et al. Making sensor networks ipv6 ready. In *Proc. 6th Int’l Conf. on Networked Embedded Sensor Systems*, Nov. 2008.
- [5] egnite GmbH. Embedded ethernet, 12 2010.
- [6] D. Gay et al. The nesC language: A holistic approach to networked embedded systems. In *Proc. ACM SIGPLAN 2003 Conf. on Programming language design and implementation*, pages 1–11, June 2003.
- [7] D. M. German. Mining cvs repositories, the softchange experience. In *Proc. 2004 Int’l working Conf. on Mining software repositories*, 2004.
- [8] O. Gnawali et al. Collection tree protocol. In *Proc. 7th Int’l Conf. on Embedded networked sensor systems*, 2009.
- [9] G. Gousios et al. Measuring developer contribution from software repository data. In *Proc. 2008 Int’l working Conf. on Mining software repositories*, pages 129–132, 2008.
- [10] A. Hassan et al. Guest editor’s introduction: Special issue on mining software repositories. *IEEE Trans. Softw. Eng.*, 31(6):426 – 428, 2005.
- [11] L. Hattori et al. On the nature of commits. In *Proc. 23rd IEEE/ACM Int’l Conf. on Automated Software Engineering*, pages 63–71, 2008.
- [12] J. Hill et al. System architecture directions for networked sensors. volume 35, pages 93–104, November 2000.
- [13] A. Hindle et al. What do large commits tell us?: a taxonomical study of large commits. In *Proc. 2008 Int’l working Conf. on Mining software repositories*, pages 99–108, 2008.
- [14] P. Levis et al. T2: A second generation os for embedded sensor networks. Technical Report TKN-05-007, Telecommunication Networks Group, TU Berlin, Nov. 2005.
- [15] M. Marin et al. Identifying aspects using fan-in analysis. In *Proc. 11th Working Conf. on Reverse Engineering*, pages 132–141, 2004.
- [16] K. Mens et al. Pitfalls in aspect mining. In *Proc. 15th Working Conf. on Reverse Engineering*, pages 113–122, 2008.
- [17] C. Thomson et al. Correctness of data mined from cvs. In *Proc. 2008 Int’l working Conf. on Mining software repositories*, pages 117–120, 2008.
- [18] T. Uno et al. Lcm ver.2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *Proc. 1st Int’l workshop on open source data mining: frequent pattern mining implementations*, 2004.
- [19] S. Walton et al. Resource management aspects for sensor network software. In *Proc. 4th workshop on Programming languages and operating systems*, pages 5:1–5:5, 2007.
- [20] T. Zimmermann et al. Mining version histories to guide software changes. In *Proc. 26th Int’l Conf. on Software Engineering*, pages 563–572, 2004.