# Resource Management Aspects for Sensor Network Software

Sean Walton Eric Eide
University of Utah, School of Computing
{swalton, eeide}@cs.utah.edu

## Abstract

The software that runs on a typical wireless sensor network node must address a variety of constraints that are imposed by its purpose and implementation platform. Examples of such constraints include real-time behavior, highly limited RAM and ROM, and other scarce resources. These constraints lead to crosscutting concerns for the implementations of sensor network software: that is, all parts of the software must be carefully written to respect its resource constraints. Neither traditional languages (such as C) nor component-based languages (such as nesC) for implementing sensor network software allow programmers to deal with crosscutting resource constraints in a modular fashion.

In this paper we describe *Aspect nesC (ANesC)*, a language we are now implementing to help programmers modularize the implementations of crosscutting concerns within sensor network software. Aspect nesC extends nesC, a component-based dialect of C, with constructs for aspect-oriented programming. In addition to combining the ideas of components and aspects in a single language, ANesC will provide specific and novel constructs for resource-management concerns. For instance, pointcuts can identify program points at which the run-time stack is about to be exhausted or a real-time deadline has been missed. Corrective actions can be associated with these points via "advice." A primary task of the Aspect nesC compiler is to implement such resource-focused aspects in an efficient manner.

## 1. Introduction

A wireless sensor network consists of inter-communicating embedded devices (called *motes*) that collect, process, and distribute data gathered from their surroundings. These networks are being developed for an increasing number of applications such as monitoring natural environments [26], localizing urban events [22], and structural health monitoring [18].

Many wireless sensor networks are intended to operate for long periods of time, over relatively large physical spaces, and in places that are difficult for people to reach. Thus, the motes that make up a wireless sensor network must be energy-efficient and relatively inexpensive. As a result, typical motes are highly resource-constrained. The popular Mica2 platform, for instance, contains just 4 KB of RAM and 128 KB of flash memory.

Such resource constraints have a strong impact on the development of sensor network software [13]. Because resources such as memory can be easily exhausted through careless coding, applications must be designed with resource conservation as a primary concern. For example, many sensor network programs must be written in ways that deal with the following issues (and others):

- **Limited stack space.** The execution stack space across 8- and 16-bit microcontroller platforms varies widely in size and flexibility—from very small (say, 100-byte) stacks at fixed addresses to stacks that can fill available RAM and be located anywhere. Stack overflows are not detected in hardware by most mote platforms and can occur any time stack is consumed: e.g., by a procedure call, auto variable, or interrupt.
- **Real-time deadlines.** Many sensor network applications must behave in a timely manner for interacting with hardware components (e.g., sensors), implementing application protocols (e.g., heartbeat signals), or both. Ensuring real-time behaviors can be difficult, especially when parts of the overall application are scheduled dynamically or involve interrupts. Less predictable than stack issues, deadlines can occur at any time during task execution.
- **API-managed resources.** Some resources, such as packet buffers, may be created and managed by OS-like or middleware-like software that underlies an application. Like bare physical resources, such "logical" resources are often limited and must be used conservatively by an application.

An essential part of designing and implementing a sensor network application, therefore, is to address resource management and exhaustion. An undetected resource error—e.g., a stack overflow or a null pointer returned from a `malloc`-like routine—can lead to seemingly "random" misbehaviors and crashes that can greatly reduce the utility of a network as a whole. In some cases, it may be acceptable to deal with resource errors merely through detection and coarse remedies such as whole-node reboots. This suffices when errors are sufficiently rare and no important state is lost, but not all applications meet these criteria. In some of these other cases, it may be necessary to eliminate resource errors at compile time, through techniques such as stack analysis [21]. Static techniques, however, are imprecise: some programs that are truly error-free at run time may be judged to contain potential errors. For this reason, it is common to combine static analysis with compiler-inserted dynamic checks to ensure that resource properties, such as type and memory safety, hold [20]. If a dynamic test fails at run time, some remedial action is taken: e.g., halting or rebooting the mote.

When halting or rebooting an entire node is unacceptable, a programmer is faced with two tasks. First, he or she must specify the behavior that should occur when a resource error is imminent or has occurred. Second, the programmer must implement that

This material is based upon work supported by the National Science Foundation under Grant No. 0410285.



The definitive version was published in *Proceedings of the Fourth Workshop on Programming Languages and Operating Systems (PLOS)*, Stevenson, Washington, USA, Oct. 2007, http://doi.acm.org/10.1145/1376789.1376796

behavior in a complete and consistent fashion throughout his or her application. Although tools exist to help programmers deal with issues such as protocol failures [11], to our knowledge, there are no tools or languages for sensor network programming that are designed to help people specify the treatment of resource errors.

To address this gap, we are now designing and implementing *Aspect nesC (ANesC)*, a new language for sensor network programming. Our language is based on nesC [10], a popular language for sensor network programming that adds component-oriented constructs to C. Aspect nesC adds constructs for aspect-oriented programming (AOP) to nesC. This is motivated by the insight that resource management policies are concerns that "crosscut" the many components that constitute a nesC application. The aspect-oriented features of ANesC are based on those found in general-purpose AOP languages such as AspectJ [14] and AspectC++ [23]. ANesC adapts these features to fit the component-oriented style of nesC.

More significantly, ANesC provides unique constructs that are designed to help programmers modularize the implementations of resource-management concerns within sensor network applications. For example, special pointcut designators will identify program points at which a stack overflow is imminent or a real-time deadline has been missed. Other constructs will allow programmers pick out interesting events on "logical" resources, such as the packet buffers described previously. Through AOP, a programmer will be able to specify—in a modular fashion—the actions to be taken when resource-related events occur. Because AOP allows programmers to refer to program context (e.g., the currently executing component), the responses to resource events can differ according to program state. Alternatively, a programmer can declare that statically decidable events such as stack overflows should be completely ruled out at compile time, using well-known AOP idioms such as "`declare error`" [14]. In all cases, it is the task of the ANesC compiler to implement the programmer's specifications in ways that are precise and efficient at run time.

The ultimate goal of Aspect nesC is to help programmers "disentangle" the implementations of resource-related concerns—and other crosscutting concerns—from the components that make up their sensor network applications. The design and implementation of the ANesC language and compiler are currently in development. This paper describes our work in progress and explains ANesC concepts in the context of a hypothetical sensor network application.

## 2. Background

Our Aspect nesC language adds aspect-oriented programming constructs to nesC, which itself is a component-oriented extension to C. We briefly review nesC and AOP below.

### 2.1 nesC

nesC [9, 10] is the implementation language for TinyOS [15], a popular, open-source, and component-based software platform for sensor network applications. The nesC language is a componentized flavor of C and introduces three language constructs for designing components: modules, interfaces, and configurations.

- A *module* is a component that is implemented by nesC code. Its body is made up of function and variable definitions, and these are encapsulated within (i.e., private to) the module unless they are explicitly exported via an interface.
- An *interface* describes a bidirectional connection between components. It names a set of functions that must be implemented by any component that *provides* the interface, as well as a set of functions that must be implemented by any component that *uses* the interface. The former are called *commands*, and the latter are called *events*. This follows the idea that the user of an interface invokes commands upon and later receives events from the provider.
- A *configuration* is a component that is implemented by connecting, or *wiring*, a set of other components. The components within a configuration are explicitly connected to one another via their used and provided interfaces. The inner components' interfaces can also be wired to the interfaces of the configuration itself. All wiring specifications are static; they cannot change at run time.

A complete nesC application is a configuration of components. Typically, a configuration contains a relatively small number of application-specific components—"business logic"—and many components that are directly reused from the library provided by TinyOS. The nesC compiler translates the entire nesC application (including the TinyOS components) into a single C file, which is then compiled by the GNU C compiler [10]. This strategy supports static error checking and whole-system optimization.

At run time, a nesC application operates as a collection of *tasks* and interrupt handlers. Tasks are invoked by the TinyOS scheduler and do not preempt each other: this run-to-completion model helps avoid multitasking errors. An interrupt handler, however, is invoked by a hardware interrupt and may preempt a task (or other handler). The nesC language provides features, such as `atomic` blocks, for managing concurrency between tasks and interrupt handlers [10].

### 2.2 Aspect-oriented programming

The goal of aspect-oriented programming (AOP) is to help programmers modularize the implementations of crosscutting concerns (CCCs). Briefly stated, a crosscutting concern is one that affects or is affected by many parts of a program's implementation. In a procedural or object-oriented language, the code of a CCC would be "scattered"—spread over many program points. An aspect-oriented language, in contrast, provides constructs to allow such a concern to be written as a cohesive unit of code.

Many AOP languages, including Aspect nesC, are designed around the notions of join points and advice. A *join point* is a point in a program's execution at which new behavior can be introduced. New behavior is called *advice*: it can augment, modify, or replace a program's existing behavior. Join points are selected according to syntactic and semantic predicates (*designators*) over a program's code and behavior, and a set of join points is called a *pointcut*. Many AOP languages also support *inter-type declarations* (ITDs), which allow new fields and methods to be added to existing datatypes in the style of open classes [7].

All of these constructs can be grouped into an *aspect*, which is a modular container of crosscutting behavior. The process of incorporating an aspect into the compiled representation of a program is called *weaving*.

## 3. Aspect nesC (ANesC)

TinyOS and nesC open new opportunities to use more advanced technologies in embedded systems, but there are still stumbling blocks that nesC cannot adequately address. In particular, key constraints and concerns still crosscut functions, components, and programs. Aspect nesC seeks to address such crosscutting concerns through aspect-oriented programming.

### 3.1 A sensor network application

To make our discussion of ANesC more concrete, we first describe some of the implementation problems that might be faced in a hypothetical wireless sensor network application for detecting and tracking tornados.

In the 1970s, researchers discovered that natural events such as tornados have infrasound signatures. Scientists have successfully used infrasound detectors in the field, but full-featured computers still accompany the tests, and since the sensors use sound, nearby human-generated noise may affect test results [3, 6].

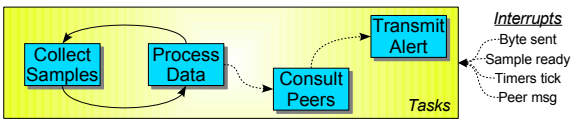


**Figure 1.** The basic device collects information and processes the information. If the device finds a match, it notifies central control.

In contrast, a large-scale wireless sensor network could operate autonomously—detecting events with minimal human interaction—and thus facilitate research. Such a network could be made from geographically distributed stations with infrasound detectors and radios. Many stations would need to be deployed, so scientists might choose to place a mote in each station, instead of full-fledged PC, in order to reduce cost and energy consumption.

The use of motes in this application leads to difficult resource-management challenges in the system's software. Each site must gather and process comparably large sound samples while coordinating with other devices and a control center. Once a sample is obtained, it must be analyzed and then compared with results stored in a database of "interesting" acoustic signatures. The best diagnosis is through sound frequencies thus requiring a Discrete Fourier Transform (DFT) to convert the sampled data, and one computationally efficient implementation is called a Fast Fourier Transform (FFT). Nonetheless, FFT and database searches are costly in time and space. Finally, when a sample is found to be interesting, the mote would consult its nearby neighbors for confirmation and then transmit its data to a control center. The overall architecture of the detector application is shown in Figure 1.

This application is naturally resource-constrained in several ways. Data gathering and analysis tasks will contend for memory and CPU: however, these tasks do not necessarily need memory at the same time, so sharing memory between these components may reduce overall memory requirements. This would mean that data collection could not run while analysis is in progress, however, leading to a real-time deadline for the analysis task. The stack is also at risk. The system designer must typically choose a stack size at compile time, but two classes of computation can consume more stack space that a system designer can (or chooses to) allocate: the FFT and interrupt servicing. FFT is naturally recursive, but the depth is bounded by the sample size. Similarly, interrupts may not be statically bounded, since a mote can be interrupted by a number of sources concurrently.

A solution is needed to help the programmer manage these resource constraints in a system-wide fashion. Because of AOP's intrinsic ability to target CCCs, we see it as a natural approach.

### 3.2 Aspects for resource management

An aspect in ANesC is written as a special sort of component, as shown in upcoming figures. ANesC implements a "standard" set of AOP constructs like those found in AspectJ and AspectC++. This includes common pointcut designators (e.g., `call`, `args`, and `within`), various types of advice (`before`, `after`, and `around`), and ITDs. ANesC extends these constructs with new features for nesC and for handling specific resource-management concerns. These special features fall into three main categories.

- First, ANesC's aspect language works with nesC's extensions to C: components, interfaces, commands, events, and tasks. Using these extensions, ANesC pointcut designators can capture join points within a certain component's implementation of a particular interface, for example.
- Second, an ANesC aspect can manipulate the wiring within a configuration. (Recall that a configuration is a component that is made by instantiating and connecting other components.) By "advising" the component graph within a configuration, an aspect can redirect wirings and add new components. Resource-management aspects can use this feature, for example, to introduce components that implement desired policies.
- Third, ANesC defines new and specialized pointcut designators to capture resource-related program events, e.g., "deadline reached" and "imminent stack overflow." Such designators allow programmers to express their intent clearly. Moreover, the ANesC compiler is responsible for locating the relevant join points, and it can perform static analyses to improve precision and produce optimized, woven code.

```
aspect WatchRealtimeDeadline {
} implementation {
    advice after(): task_deadline_reached() {
        //--- Change FFT accuracy, reducing calculations
    }
    advice before(): task_completion() {
        //- If ends before deadline
        //-     Increase analysis accuracy
        //- If accuracy tuning was visited before
        //-     Revert accuracy
    }
    ...
}
```

**Figure 2.** Adaptation for a real-time deadline. If the analysis deadline is hit before task completion, turn the analysis accuracy down. Readjust as deadlines are met, but try to avoid "bouncing."

***Adapting for real-time deadlines.*** A nesC application runs as a collection of non-preemptive tasks. This scheduling strategy can make it difficult to build systems that meet real-time deadlines, especially when tasks have variable (e.g., data-dependent) processing times. A task that overruns its deadline cannot simply be aborted: that would leave locks held, resources allocated, and produce incorrect results in general. An alternative is for tasks to be self-monitoring. When a task detects that it has overrun, it cleans up safely and terminates as quickly as possible. When coupled with a mechanism for adjusting the workload of future tasks, this strategy allows a system to adapt to meet soft real-time deadlines.

To implement self-monitoring in plain nesC, a programmer would need to insert—by hand—deadline checks and cleanup steps at many places in a task's code. This is tedious, error-prone, and "tangles" the task functionality with the implementation of the real-time concern. ANesC offers a better alternative: implement the self-monitoring code as an aspect, as illustrated in Figure 2.

In ANesC, a special pointcut designator can capture task deadline events. The ANesC compiler is responsible for inserting deadline checks where they must occur. (It can implement various strategies for periodic checks, which we do not describe here.) When an overrun is detected, programmer-written advice can free resources and locks, and terminate the task safely. The same aspect can encapsulate the workload adaptation strategy as well. Figure 2 shows this for the analysis task in our tornado application. If the analysis runs too long (impacting the rate of the sample-collection task), advice throttles back accuracy and thus frees CPU time. If more CPU is made available, precision can be carefully increased.

***Avoiding stack overflow.*** Impending stack overflow is tricky, both in detection and in resolution. As described previously, in some sensor network applications, it may not be practical to statically eliminate the possibility of stack overflow due to resource or analysis limits. In such systems, we need to address overflows at run time. The desire is to detect overflow *before* it actually occurs, and

```
aspect PreventStackOverflow {
} implementation {
    advice inline after(): stack_overflow_imminent() &&
      withintask(task void process_data(void)) {
        //- Report the current results of processing
        //- Stop task and release buffers
    }
}
```

**Figure 3.** Catching an imminent stack overflow. An application can take corrective action before the stack overflow occurs.

then deal with the event gracefully. Static information from the compiled program (object files) is needed to implement overflow checks accurately and efficiently.

Figure 3 outlines how the tornado detector could respond to a stack overflow in the analysis task by using the intermediate results and halting the task. The analysis' stack demand can be predicted by static analysis, because the FFT consumes $O(\log_2 N)$ stack frames for an $N$-sample buffer. Suppose, however, that due to resource constraints, the size of the system's stack segment is variable. For instance, when the system is put in a (very rare) diagnostic mode, part of the stack segment is repurposed to hold a log. When this occurs, there is not enough stack space for the normal FFT execution path. Fortunately, if the stack fills up, all is not lost: DFTs become more precise with each pass, but even the intermediate results are useful. If the stack space is unexpectedly exhausted—an event detected by a custom ANesC pointcut designator—advice can deliver intermediate results as the final output.

The implementation of the "imminent stack overflow" designator is platform-specific, depending on the flexibility and size of the stack. Furthermore, the location of stack checks in the woven program depends on information from the C compiler and static analysis tools. Resolution is also tricky: how can a programmer advise a join point that does not have enough stack for another section of code? For this reason, advice at stack-full join points must be written carefully (and checked by the ANesC compiler). We expect that common recovery advice will involve an exception-like mechanism for unwinding to an appropriate program point.[1] Other strategies such as stack compression are possible in some cases; exploring these is part of our current research.

***Managing memory buffers.*** Having limited RAM naturally encourages the use of memory pools [13], or requiring a certain amount of trust so that a handed-over buffer is not reused, or careful scheduling of multiple tasks that share a block of memory. The best strategy for memory management within a nesC component is often context-specific: that is, not a property of a component itself, but a property of how a component is *used* in a complete application. Moreover, once a particular strategy is chosen, it must be implemented in a coordinated manner over many individual components. AOP can simplify this task so that a programmer can focus on resource strategies, replacing one with another without having to rewrite large sections of program code.

Consider our hypothetical tornado detector with limited RAM. Each component needs a certain amount of RAM to receive or process samples, collect frequency results, or transmit a message. Having well-defined places where each component expresses a need for memory—i.e., join points—makes it possible for an aspect to introduce a memory strategy in a straightforward way. In our application, an appropriate strategy would be for components to allocate buffers dynamically from a single memory pool.

[1] Jacobsen et al. provide a catch/throw-like mechanism in ACC [12]. However, "ACC-exception" handling has several limitations and does not implement state unwinding.

The system has two characteristics that enable a memory-pool strategy. First, the sampling rate for the infrasound range (1–20 Hz) is a very low 40 samples/sec. Thus, a single sample may not need to occupy most of RAM, and processing can proceed while collecting data without much regard for pool underflows. Second, DFTs split the sample array in half after each pass, turning the array into an ordered tree. As the analysis task makes a pass through the tree in an in-order traversal, parts of the sample buffer can be incrementally released to be refilled by the data collector.

When there is need to report a sighting, ANesC advice can redistribute buffers for a message to the home station. Advice can change the sample size parameters for the data collector and analyzer, thereby freeing memory for the reporter task.

### 3.3 Discussion

Applying high-level techniques like AOP to sensor network programming reasonably raises concerns about impacts on performance and memory footprints. Each level of language abstraction often increases binary footprint by some percentage, and certainly, dynamic deadline checks and stack checks will invariably increase code footprint a few bytes per test. However, the nature of AOP—being able to modularize crosscutting concerns—offers the potential to reduce code footprint overall by modularizing code that is currently duplicated across components.

The design of ANesC from the beginning has focused on code and data footprint impact. This is in contrast with some other AOP languages, which offer features that are expensive in space and time. For example, our analysis of an earlier version of ACC [12] discovered that ACC added several method call layers, and even marshaled and unmarshaled parameters. Clearly, such an approach would not be conducive to an embedded system in which the stack is shallow and memory and CPU are scarce.

Finally, an important benefit of adding AOP functionality to nesC is the ability to advise a sensor network application *and* its operating system, TinyOS, at the same time. This is possible because the application and OS are compiled together as a single program. This opportunity is not readily available to most other AOP languages.

## 4. Related Work

Our ANesC language is part of a family of efforts to extend C and C-based languages with AOP constructs. These languages include AspectC [8], ACC [12], and AspectC++ [23]. Like ANesC, most of these languages have been applied to modularizing concerns within embedded and/or systems code. Unlike our language, however, these previous languages have been designed for "general-purpose" AOP, meaning that they are not tailored to addressing specific concerns. ANesC, on the other hand, is designed with a specific problem domain in mind: improving the implementations of resource-management concerns on small devices. To this end, it extends general-purpose AOP constructs with novel constructs that pick out resource-related program points.

Applying general-purpose AOP to embedded systems continues to be an active area of research, especially for investigations into software product lines for embedded software. For instance, Afonso et al. discuss the use of AspectC++ for the BOSS embedded operating system [1], and Pukall et al. and Tesanovic explain their efforts in using other AOP languages to develop product lines of embedded systems [19, 25]. In separate papers, Beuche and Spinczyk describe their product-line systems based on the AspectC++ *Pure::Consul* library [5, 23]. Lohmann and Spinczyk describe their implementation of a weather-station application on a microcontroller platform and present the tools that they use including *Pure::Variants*, a variant-management system [16]. We believe that the focus of much of this work is complementary to our own.

One can easily imagine resource-management aspects being significant elements of an embedded system product line.

Like ourselves, other researchers have applied language techniques to address particular concerns of sensor network programming. Gummadi et al., for example, propose the use of "macroprogramming" to create a distributed program composed of multiple firmware images acting in concert to perform specified tasks [11]. Regiment and COSMOS use this concept to manage the crosscutting concern of embedded node failures and recovery [2, 17]. Their work is similar to ours in that they use a language to modularize a crosscutting concern. Our approach differs from theirs, however, in that ANesC combines its features for problem-specific AOP with well-known constructs for general-purpose AOP.

Over the years, languages and systems such as Ada, Esterel [4], Java Card [24], and others have been developed to improve the design and implementation of embedded software for microcontroller-class devices. Despite the many desirable features of these languages, most embedded software today continues to be written in the C family of languages. The nesC language provides evolutionary improvements for C-based sensor network programming. Our hope is that ANesC will enable a similar evolutionary approach for the management of crosscutting concerns in the sensor network domain.

## 5. Conclusion

We have described Aspect nesC (ANesC), our new language that seeks to improve programmers' control of resource management concerns in sensor network software. Resource management is an essential part of most sensor network software because common motes are resource-impoverished. In addition, because resource constraints generally influence many or all parts of an application, resource management is a classic example of a crosscutting concern. ANesC is designed to help programmers manage such crosscutting concerns through aspect-oriented programming. We have described how ANesC adds aspects to the component-oriented features of the existing nesC language. In addition, we have described how ANesC will provide novel features for modularizing the implementations of resource-management concerns in particular, and we have discussed ANesC's application through a hypothetical tornado-detector example. Through these language-based techniques, our goal is to improve both (1) the behavior and resilience of sensor network applications in the face of resource shortages, and (2) the flexibility of such software to changes in resource concerns. Both the ANesC language and its compiler are currently in development.

## Acknowledgments

We thank David Gay for his assistance with the inner workings of the nesC compiler, and we thank John Regehr for his insightful comments on drafts of this paper.

## References

[1] F. Afonso, C. Silva, S. Montenegro, and A. Tavares. Applying aspects to a real-time embedded operating system. In *Proc. of ACP4IS*, Vancouver, BC, Canada, Mar. 2007.

[2] A. Awan, S. Jagannathan, and A. Grama. Macroprogramming heterogeneous sensor networks using COSMOS. In *Proc. of EuroSys*, pages 159–172, Lisbon, Portugal, Mar. 2007.

[3] A. J. Bedard Jr. and T. M. Georges. Atmospheric infrasound. *Physics Today*, Mar. 2000.

[4] G. Berry. The foundations of Esterel. In *Proof, language, and interaction: essays in honour of Robin Milner*, pages 425–454. MIT Press, 2001.

[5] D. Beuche and O. Spinczyk. Variant management for embedded software product lines with Pure::Consul and AspectC++. In *OOPSLA Companion*, pages 108–109, Anaheim, CA, USA, Oct. 2003. Demonstration abstract.

[6] H. B. Bluestein. A history of severe-storm-intercept field programs. *Weather and Forecasting*, 14(4):558–577, Aug. 1999.

[7] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proc. of OOPSLA*, pages 130–145, Minneapolis, MN, USA, Oct. 2000.

[8] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proc. of ESEC/FSE*, pages 88–98, Vienna, Austria, Sept. 2001.

[9] D. Gay, P. Levis, D. Culler, and E. Brewer. nesC 1.2 language reference manual. `http://sourceforge.net/projects/nescc`, 2005.

[10] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of PLDI*, pages 1–11, San Diego, CA, USA, June 2003.

[11] R. Gummadi, N. Kothari, T. Millstein, and R. Govindan. Declarative failure recovery for sensor networks. In *Proc. of AOSD*, pages 173–184, Vancouver, BC, Canada, Mar. 2007.

[12] H.-A. Jacobsen. AspeCt oriented C compiler. `http://research.msrg.utoronto.ca/ACC`, 2007.

[13] K. Klues et al. Dynamic resource management in a static network operating system. Technical Report WUCSE–2006–56, Washington University in St. Louis, Oct. 2006.

[14] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Company, 2003.

[15] P. Levis. TinyOS programming. `http://csl.stanford.edu/~pal/pubs/tinyos-programming.pdf`, 2006.

[16] D. Lohmann and O. Spinczyk. Developing embedded software product lines with AspectC++. In *OOPSLA Companion*, pages 740–742, Portland, OR, USA, Oct. 2006. Demonstration abstract.

[17] R. Newton, G. Morrisett, and M. Welsh. The Regiment macroprogramming system. In *Proc. of IPSN*, pages 489–498, Cambridge, MA, USA, Apr. 2007.

[18] J. Paek, K. Chintalapudi, R. Govindan, J. Caffrey, and S. Masri. A wireless sensor network for structural health monitoring: Performance and experience. In *Proc. of EmNetS-II*, pages 1–10, Sydney, Australia, May 2005.

[19] M. Pukall, T. Leich, M. Kuhlemann, and M. Rosenmueller. Highly configurable transaction management for embedded systems. In *Proc. of ACP4IS*, Vancouver, BC, Canada, Mar. 2007.

[20] J. Regehr, N. Cooprider, W. Archer, and E. Eide. Efficient type and memory safety for tiny embedded systems. In *Proc. of PLOS*, San Jose, CA, Oct. 2006.

[21] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. *ACM Transactions on Embedded Computing Systems*, 4(4):751–778, Nov. 2005.

[22] G. Simon et al. Sensor network-based countersniper system. In *Proc. of SenSys*, pages 1–12, Baltimore, MD, Nov. 2004.

[23] O. Spinczyk, D. Lohmann, and M. Urban. Advances in AOP with AspectC++. In *Proc. of SoMeT*, Tokyo, Japan, Sept. 2005.

[24] Sun Microsystems. Java Card Platform Spec. 2.2.2, Mar. 2006. `http://java.sun.com/products/javacard/specs.html`.

[25] A. Tesanovic. Evolving embedded product lines: Opportunities for aspects. In *Proc. of ACP4IS*, Vancouver, BC, Canada, Mar. 2007.

[26] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proc. of OSDI*, pages 381–396, Seattle, WA, Nov. 2006.