Software Analysis For Wireless Sensor Networks

Roy Shea

Computer Science Department,

University of Los Angeles, California,

roy@cs.ucla.edu

February 20, 2007

Contents

1 Lessons From An I2C Driver **3**

2 Thesis Overview **5**

3 Background **6**

3.1 Embedded Sensor Networks 6

3.2 Embedded Systems 10

3.3 Programming Languages 12

4 Past Work **14**

4.1 SOS 15

4.2 Lighthouse 16

4.3 Understanding the Basic Data Flow 18

5 Future Work **20**

5.1 Joining Static Analysis and State Machine Specifications 20

5.2 New Directions for Lighthouse 22

6 Conclusions **25**

1 Lessons From An I2C Driver

My first exposure to low layer hardware drivers occurred early in the SOS [21] operating system's life. I wanted to add features to make the operating system for 8-bit devices more usable to other members of the Networked and Embedded Sensing Laboratory. Thinking to extend SOS support of the MTS310 sensor board, I looked towards the TinyOS [14] magnetometer driver. The magnetometer on the MTS310 has a narrow sensitivity range that is adjusted with the help of an external resistor. A simple algorithm in the TinyOS driver adjusts a digital potentiometer to calibrate the magnetometer for the surrounding magnetic field. While the algorithm was simple, I found the code to adjust the digital potentiometer baffling. Manual bit banging routines used a mix of explicit assembly `nop`s and confusing state transitions. I gladly redirected my attention to the SOS dynamic memory subsystem.

A year later I solved the mystery of the TinyOS potentiometer driver as I began developing an I2C driver for SOS. A group using SOS wanted better support for the I2C bus protocol used to connect and control simple devices. Examining TinyOS drivers to bootstrap development for SOS, I realized that the mysterious bit banging controlling the potentiometer was a manually timed I2C driver. Data sheets for the ATmega128L, the microcontroller running SOS, revealed hardware support for I2C. This hardware support handles timing and uses interrupts so that the microcontroller can perform other tasks once an I2C transaction has been initiated. I decided that the SOS I2C driver would use this hardware support and began development.

This driver development proved to be more difficult than I anticipated. The driver needed to support both master mode, where the device initiates and controls the I2C transaction, and slave mode, where the device responds to I2C requests from another devices. These two modes resulted in a driver with many states and complicated interactions to SOS. Ad hoc testing of the driver revealed bugs common to pointer usage within C and bugs resulting from improper interfacing of the driver to SOS. Difficult to pin down bugs raised concerns about the potential for erroneous sequences of state transitions within the I2C driver. But further testing eventually provided enough confidence to warrant a preliminary release.

More robustness questions arose six months later as the driver expanded to include multi-master support. Multi-master I2C allows multiple devices on the bus to act as both master and slave. Master devices engage in an aberration protocol during a transaction. Users requested multi-master I2C

support to enable communication between groups of processors running SOS. Again the complexity of the driver and its interactions with SOS grew. New bugs were found in poorly tested and unexpected control paths between SOS and the driver. Eventually the driver reached a stage of stability where the benefits of the driver outweighed the uncertainty about the driver's robustness.

I2C driver development reached a pivotal point a full year later when preparations to demonstrate imaging hardware were hindered by SOS frequently hanging. SOS was running on both a Mica2 mote and on the coprocessor of an attached camera. The I2C bus was used to transfer SOS programs from the Mica2 mote to the imaging hardware and to transfer processed photos from the imaging hardware to the Mica2 mote. After a great deal of searching, since the bug had not been pinned down to a subsystem of SOS, a careful review of the I2C protocol specification revealed the problem. Masters on a multi-master I2C bus must always transmit the same number of data segments and each segment must be the same length. SOS violated this requirement causing the I2C hardware to enter an illegal state and hang.

Problems encountered during I2C driver development embody challenges faced by embedded sensor network programmers. Initial problems stemmed from pointer problems common to C programs. Later versions suffered from bugs caused by the complex interface between the I2C driver and the higher level management system provided by SOS. A particularly nasty bug stemmed from a misunderstanding of the underlying hardware's capabilities.

Techniques that solve these same problems in other domains are difficult to apply to sensor networks. Memory safety techniques developed for C often depend on large metadata structures that do not fit well in the limited RAM on sensor nodes. The inherently event based nature of sensor networks makes interprocedural analysis of sensor network programs very challenging. Good specification languages have not yet found a receptive audience in the sensor network community.

My dissertation work focuses on solving these problems faced by sensor network developers. Over the past two years I have developed an analysis framework that allows traditional static analysis be used in the sensor network domain. I plan to further develop this framework and explore its capability to create more robust sensor networks.

This prospectus describes my approach to accomplish that goal. Section 2 provides a brief overview of my proposed dissertation work. This is followed in section 3 by a detailed review of particularly relevant works from the sensor network community, the embedded systems community, and the general programing languages community. Section 4 reviews my past work with sensor

networks and section 5 explores the future work I envision for this research. The paper finishes with a brief summary in section 6.

2 Thesis Overview

Developing sensor network programs is difficult. Current research projects attempt to alleviate these difficulties in different ways. My dissertation will bring static analysis and verification to the embedded sensor network domain.

Traditional static analysis often uses modular checking to verify a subsystem in isolation, while still reasoning about the entire system. Sensor networks, with their physical ties to the environment, are reactive in nature. Unfortunately, many benefits of modular analysis are lost in reactive systems. This loss results from state space pollution caused by reactive events that, from the perspective of the analysis, may fire at any time.

My work over the past two years explores a slice of this dilemma. The work resulted in a framework that, when completed within the next two months, will vertically integrate static verification with higher level specifications. This integration allows modular static analysis to proceed with a more limited state space based on assumptions coming from the higher level specification. Load time and run time checks are added to the system to insure that assumptions from the specification are properly followed.

Dynamic memory management has driven my work up to this point. Incorrect usage of dynamic memory, and potentially other resources requiring an exclusive ownership mode, is detected at compile time by the analysis framework.

My thesis for this work is that:

> A union between static analysis and higher order specifications is critical for analysis and verification work in the event based embedded sensor network domain.

My dissertation will explore how Lighthouse can better implement this thesis while applying it to unsolved problems within sensor networks.

3 Background

This prospectus pulls ideas from across the systems domain to overcome challenges faced by sensor network developers. In this section I explore important research from sensor networking, embedded systems, and programming languages.

3.1 Embedded Sensor Networks

The field of embedded sensor networking has evolved a great deal over the past decade. Early research envisioned a field dominated by in network processing [12, 8]. Deployment experience soon displayed many challenges of the domain, such as unstable radio channels [40, 41, 6], that increase software complexity. Attempts to manage this complexity drives the development of more robust embedded sensor network systems.

3.1.1 System Design

Improved system design is one way that researchers control the effects of complex and potentially buggy sensor network software running on motes with microcontrollers.

Designed for 8-bit microcontrollers, the t-kernel [18] creates a robust system through advanced kernel techniques. Operating system protection is implemented via assembly rewriting of user code pages loaded into the running system. Virtual memory in the t-kernel provides fast access to an untrusted local stack and restricted access to a heap. The virtualization of data memory provides significant protection against rouge process that would otherwise corrupt kernel data structures.

The SOS operating system [21] implements a static kernel that provides a wide range of services to user applications written in C and offers limited protection through kernel features. A memory manager provides basic book keeping to track the memory allocated by each user application. Applications in SOS can be dynamically added to, updated on, and removed from nodes. To better support this functionality the kernel intervenes when an application attempts to use a service provided by another application but that is not available on the node.

Early sensor network systems developers recognized the need to increase system robustness. TinyOS with the help of nesC [14] provides significant structure to sensor network developers. The wiring language used in nesC encourages developers to develop interfaces that facilitate reuse of software. The nesC compiler examines all code reachable from interrupt handlers and warns users of

potential race conditions. These features provide a base level of robustness to TinyOS applications.

T2 [29], a major revision of TinyOS, further illustrates how sensor network systems are evolving to be more robust. T2 incorporates four basic principles to increase system robustness and maintainability. Telescoping abstractions provide a framework for developing hardware interfaces that span from restricted and robust to expressive but potentially dangerous. Partial virtualization provides a layer of indirection to present users a view of exclusive access to resources that are actually shared. T2 continues to use static allocation and bindings between components to facilitate compile time verification. Finally, T2 uses service distributions to provide simplified interfaces to sets of commonly used services.

Virgil [39] uses expanded language features to facilitate robust sensor network application development. Virgil provides sensor network developers a statically typed language with a base set of object oriented features. The static typing helps programmers from making many mistakes common to pointer manipulation in low layer programing. An intelligent compiler framework aggressively optimizes Virgil programs for operation on 8-bit systems.

These approaches interact with my verification work in different ways. Kernel support for more robust applications allows my verification framework to make stronger assumptions about the underlying system. As kernel support matures, my verification framework will be able to leverage stronger assumptions into its analysis. Specialized language support, as seen in nesC and Virgil, can provide additional footholds for my verification. Both nesC and Virgil already use limited forms of static analysis to provide concurrency and type safety to programs. Their specialized language features may prove to be more amenable to verification by external analysis than the C code that my framework has focused upon up to this point.

3.1.2 Simplification

Current research also looks for better abstractions to simplify sensor network design. The Tenet [17] architecture is a notable example. Tenet argues that the cost and complexity of implementing application logic and data fusion on motes outweighs the performance gains. In light of this observation it proposes a tiered network architecture with:

- Asymmetric communication where master tier nodes task motes.
- A routing scheme in which masters address any mote, and motes simply respond.

- A task library loaded onto motes that implements the generic tasks used by masters.

Examination of this idea demonstrates that the cost of the architectural change is minimal.

My analysis framework pushes on the cost and complexity trade offs described in works aiming to simplify mote code in sensor networks. Stronger guarantees about the correctness of mote code allow developers to push a little more functionality onto motes. For example, stronger guarantees from my analysis framework will allow more complex tasks in the Tenet framework.

3.1.3 Threads and State Machines

The event driven nature of sensor networks often leads to complex state machines within user code that are difficult to understand and error prone. These state machines are often implemented ad hoc using nested switched statements or many small functions called by an event dispatcher. Illegal state machine transitions are not always monitored against and can lead to run time errors when the environment behaves in an unexpected manner. Non-preemptive operation common to sensor network mote devices exacerbate this problem by requiring long operations to be manually split into multiple tasks. Some sensor network research looks to relive these problems through improved program structure.

Protothreads [11] propose one solution to this problem. This work is based on the observation that many programs use a mostly linear chain of state transitions. Protothreads introduce conditional waits and yield points into sensor network systems. This allows a program to be written in a natural linear from, rather than broken up into independent events. This abstraction of threads is limited, as blocked protothreads do not have their context saved on the stack.

TinyThreads [30] take this idea ever further by providing a threading library for TinyOS. TinyThreads use a dedicated stack for each thread that is statically allocated based on a compile time stack analysis. Threads in the system are scheduled cooperatively. This work argues that the threading abstraction will ease the development of more complex applications.

These alternate ways of describing a program provide inspiration for the specifications used in my analysis work. Loosely structured state machines can result in an analysis having to assume that any event can fire at any time. These threading abstractions provide a stylized way to limit the valid transitions that the analysis needs to consider.

Research also exists to better understand the relationship between interrupts and threads in theses

systems. One recent paper [34] observes that, while threads are a dominant abstraction for use in large system design, embedded systems are better suited to events due to limited space for stacks. However, analysis tools for threads tend to be more developed than their event based counter parts. The paper begins to outline a source to source translations that rewrites interrupt driven software in a threaded style, allowing analysis from thread verification software.

This line of research motivates the theme of proper specification that runs through my analysis work. Often, simply describing a system in the correct manner will facilitate better analysis and increased robustness.

3.1.4 Safety

The sensor network community recognizes that increased system safety and robustness is needed. Emerging tools combine traditional analysis with new techniques specially suited for the sensor network domain.

A good example of this is the UTOS [35] work, which combines a number of separate tools to create a trusted environment for TinyOS applications. CCured [33] and custom analysis transform untrusted applications to ensure type and memory safety while optimizing performance for execution on an 8-bit microcontroller. Untrusted applications are only allowed to call into the kernel through a limited set of trusted proxy functions. The proxies manage resources and access to underlying hardware for the untrusted applications.

Other work examines specification of RAM, ROM, and stack requirements of individual TinyOS components [23]. Such a specification facilitates robust integration of components from different developers into a single application. Approximation of these properties is currently implemented by direct examination of the ELF file.

My verification work aims to refine this class of techniques. In a system such as UTOS, my analysis may be able to further reduce the overhead of instrumented code by refining the analysis preformed by CCured. Similarly, more accurate bounding of RAM, ROM, and stack usage is required if such specifications are to be of value in real systems.

3.2 Embedded Systems

Embedded sensor networking is closely tied to the more developed field of embedded systems but also displays important differences. The embedded systems field offers a well studied collection of design methodologies with strong verifiable properties. A point to understand is how sensor networks differ from more traditional embedded systems. One notable difference is in the applications targeted for each domain. Where traditional embedded systems are often used for very specialized processing tasks, sensor network nodes tend to be more general and are occasionally multi-tasked. A second notable difference arises from the researchers attracted to the respective fields. Traditional embedded systems have attracted attention mostly from electrical engineers. In contrast to this are sensor networks that have attracted a great deal of attention from software engineers who bring with them backgrounds in system and language design.

3.2.1 Design Paradigms

Data flow is a fundamental design paradigm used in embedded systems for over twenty years. Data flow describes computation as a graph of functions with links connecting inputs and outputs. A foundation class of data flow systems is synchronous data flow (SDF) [26]. In SDF the execution of a node generates a fixed number of tokens onto its output link, and a node always consumes a set number of tokens from an input link to fire. This constraint allows static analysis of an SDF that can find a static schedule for the SDF if one exists, and alert the user if the graph can not be scheduled for infinite execution with bounded buffers. Extensions to SDF, such as cyclo-static data flow [4], begin to add in more flexibility at the cost of more complex analysis.

Data flow techniques are not widely used in sensor network systems. The rigid structure of SDF is often a poor match for the more general conditional event transitions used in sensor networks. Data flow has found limited use within restricted computation models built on top of sensor network systems. Virtual machine environments like Mate [28] and design architectures like Tenet [17] create environments where tasks are executed in a style similar to data flow.

My verification work targets more general software that does not need to adhere to the restrictive data flow models. In situations where sensor network software does follow this model, my verification work can leverage the theory already established by the embedded systems community.

Kahn Process Networks (KPN) are a more general model studied within the embedded systems

domain. KPN use networks of functions that communicate through unbounded FIFO channels. This more general abstraction is a better map onto sensor network systems. Recent work on KPNs [15] describes requirements for a correct and bounded schedules. Formalisms from KPNs may find a home within my verification work as I look for better means to specify valid sequences of event firings an embedded networked system.

State machines and related variations are another design paradigm studied within traditional embedded systems. While research has examined formal properties of state machines, non-trivial systems often result in large and unwieldy state machine specifications. Statecharts [22] provided a significant improvement over simple finite state machine representations. Statecharts use XOR decomposition and AND unions of state machines to significantly reduce the number of states required to describe a system. Difficult semantics result from these additional features, primarily concerning ordering of cascading side effects that are required to be modeled as occurring "instantly". An alternative take on statecharts are hierarchical finite state machines (HFSM) [16]. In this model, finite state machines are embedded within one of several concurrency models including synchronous data flow, discrete events, or synchronous / reactive systems. Nesting within HFSMs produces state reductions similar in scale to statecharts, but with clearer semantics.

The event based nature of sensor network programs directly maps onto these formalisms and I am already applying state machines to my analysis framework. More advanced representations, such as statecharts and HFSMs, will be examined as more capable replacements to the state machines I currently use.

3.2.2 Concurrency

Concurrency can be a significant problem for sensor network applications and has long been studied in the embedded systems domain. Some embedded systems researchers argue that threading is not an appropriate abstraction for embedded systems [27, 25]. They argue that threading results in complex and non-deterministic interactions between distinct parts of the system that are difficult to understandable and difficult to verify. Timing constraints also play into concurrency problems. Some embedded systems literature [31] notes that real time operating systems is not always a good fit for a problem. This is seen in most current sensor network operating systems, which do not provide real time guarantees. However, some aspects of sensor networks that are coupled with I/O devices do require delicate timing. Attempts to uphold timing constraints increase concurrency risks

as preemption plays a greater role in the system.

My analysis work is closely tied to these concerns about concurrency. As noted in section 3.1.3, sensor network research is looking towards the threading abstraction as a means to write clearer programs. Since current threading work in sensor networks minimizes preemption, the abstractions do not have many of the problems recognized by embedded systems researchers. But if preemption is introduced to these threading abstractions or if tight timing constraints introduce more concurrency to these systems then the state space of static analysis will significantly increase. This increases the importance of clear specifications in my analysis framework.

An alternative approach to handling concurrency problems is through synchronous languages such as Esterel or Lustre [3]. This style of programming may be applicable to sensor networks that have tight timing constraints. Research is also exploring how to apply the restrictive synchronous models to more general systems [19].

My knowledge of synchronous languages is quite limited and at this time my verification work does not consider this topic. However, as my work progresses this may be an area to revisit.

3.3 Programming Languages

The programing languages community has long researched techniques to make systems more robust and more amenable to verification. Techniques range from external tools that specialize in verifying particular system properties to language extensions that specify system requirements.

3.3.1 Program Analysis and Augmentation

External analysis that examine and possibly augment a software system can increase system robustness. This is a convenient approach since it does not require programs to specially modify their design practices by rewriting programs or using new languages.

An example of this technique is CCured [33]. CCured examines and augments C source code to produce pointer type safety. The analysis classifies pointers as type safe, sequence, or dynamic. Type safe pointers are statically known to be safe and require no additional attention. Sequence pointers can be wrapped in small run time bounds checkers to monitor for buffer overruns. Dynamic pointers require extra metadata that is updated during program execution to ensure safe run time use.

A second example augments a software system with automatic pool allocation to provide strong

guarantees on alias analysis [10]. This work observes that partitioning memory into pools based on a points to analysis provide extra protection against rouge pointers, creating a more robust alias analysis. The tool uses various optimization techniques when it augments the code to reduce run time overhead.

These types of analysis directly correspond to my proposed dissertation. My work seeks to make this type of analysis more applicable to the sensor network domain by providing a more accurate analysis framework that enables reduced run time overhead.

3.3.2 Language Features

Analysis of unmodified systems is attractive when applicable, but some system properties are very difficult to reason about without language support. This observation motivates the use of custom language extensions or, in some situations, custom languages.

Vault [9] is an example of such a system, adding guards to types to specify access restrictions on variables. For example, guards can specify that operations on a variable must occur in a specific order. Vault programs are statically analyzed to ensure that value guards are satisfied at each state in a program. An application and extension to this technique appears in Sing# [13]. Sing# uses typed communication channels to make guarantees about the ownership of buffers passed over the channels.

Alias Java [1] adds type annotations to objects to describe ownership properties. Annotations can denote objects that are simply being lent to other functions, unique objects that are never shared, or owned objects whose owner can grant other functions access to. Analysis on this system use these annotations to produce strong claims about the ownership of objects and find violations of ownership assumptions.

Language augmentation is an approach that my analysis framework does not currently use. One goal of my work is to push on the limits of analyzing unmodified systems. However, an important aspect of my work is access to specifications of one form or another. Minimal changes to a language or using a custom languages can efficiently embody the specification information of interest to my analysis. This union of specification with code relieves developers from writing separate specifications and is an area that I will eventually explore.

3.3.3 Specification Languages

Languages such as SLIC [2] and Metal [7] have been developed specifically for helping in the verification of systems. SLIC facilitates the specification of temporal safety properties of APIs. These specifications are then embedded within system source code. The SLAM project combines this annotated code with a static reachability analysis to determine if the constraints of the specification hold. Metal is a similar language that allows users to declare properties that are matched against the abstract syntax tree of a program. Data flow analysis updates a global state when the abstract syntax tree matches rules in the specification. This technique allows scripting of simple analysis checks.

Specification languages are directly applicable to my analysis work. My framework already uses a simple specification language similar to SLIC to specify pre- and post- memory conditions for functions being analyzed. As this framework is expanded to handle more diverse analysis, it will be important to develop an expressive and easy to use specification language. This language may continue to be similar to the specification languages described here, or may take the form of language extensions.

3.3.4 Finding Bugs

An alternative to formal verification is automated bug identification. This technique is valuable for systems or properties that do not yield to more rigorous analysis.

Notable work in this direction [20] has been done using xgcc with the Metal specification language described above. Latter work using this same framework extends the analysis using statistical techniques to automate the formation of specifications [24].

Early invocations of my analysis framework followed these ideas by locating bugs common in the SOS operating system. However, my current analysis framework and proposed thesis focus on the stronger claims made possible by a more complete analysis.

4 Past Work

I bring to sensor network research an academic background in operating system design. This knowledge is flavored by other studies in networking, security, and artificial intelligence. My programing languages and formal verification education has been more incidental, motivated by my program analysis work over the past two years.

```
53 static const mod_header_t mod_header SOS_MODULE_HEADER = {
54   .mod_id        = SURGE_MOD_PID,
55   .state_size    = sizeof(surge_state_t),
56   .num_sub_func  = 1,
57   .num_prov_func = 0,
58   .platform_type  = HW_TYPE /* or PLATFORM_ANY */,
59   .processor_type = MCU_TYPE,
60   .code_id       = ehtons(SURGE_MOD_PID),
61   .module_handler = surge_module_handler,
62   .funct = {
63     [0] = {error_8, "Cvv0", TREE_ROUTING_PID, MOD_GET_HDR_SIZE_FID},
64   },
65 };
```

Figure 1: Old versions of memCheck verify that SOS modules include a valid `SOS_MODULE_HEADER`

4.1 SOS

My first substantial work in sensor networking was the SOS Operating System (SOS) [21] for 8-bit mote class devices. SOS was conceived of while transitioning from the Sensorware [5] mobile agent platform to TinyOS [14]. TinyOS's component model and design for 8-bit systems attracted our attention, but we wanted to maintain the ability to migrate code. A brainstorming session raised the idea of rewiring components into TinyOS at run time. Due to the static binding assumption built deep into TinyOS, we began building a new operating system to implement this idea. Out of this work came SOS.

SOS uses a static kernel to provide a core set of services to user modules. All nodes in a deployment are initialized with this static kernel. At run time, user applications made of one or more modules are loaded on top of the kernel for execution. Modules communicate with the kernel through a function jump table, and with each other through indirect function references and messaging. More details of the system are available in the SOS conference paper [21] available at `http://nesl.ee.ucla.edu/document/show/15`.

While user experience with SOS has been positive, many new developers using SOS have discovered that, unlike desktop systems, sensor nodes do not respond well to sloppy programming. Accessing invalid references or subtle memory leaks often result in fatal and difficult to track down bugs. Observations of students struggling with these types of errors motivated my transition from operating system development to bug detection.

```
112   //! Requested sensor data ready
113   case MSG_DATA_READY:
114     {
115       MsgParam* param = (MsgParam*) (msg->data);
116       int8_t hdr_size;
117       uint8_t *pkt;
  ...
122       pkt = (uint8_t*)sys_malloc(hdr_size + sizeof(SurgeMsg));
  ...
135       sys_post(TREE_ROUTING_PID,
136               MSG_SEND_PACKET,
137               hdr_size + sizeof(SurgeMsg),
138               (void*)pkt, SOS_MSG_RELEASE);
139       break;
140     }
```

Figure 2: Old versions of memCheck detected simple memory errors such as a memory leak that would occur if `SOS_MSG_RELEASE` were not included in line 138 above

4.2 Lighthouse

My analysis work began in response to simple programming and style errors that commonly appeared in SOS programs. Students in the 2005 class of EE202A were the first group to use a predecessor to Lighthouse called memCheck, a collection of simple style checkers to detect common mistakes in SOS modules. These checkers were written using the C Intermediate Language (CIL) [32]. Checks included:

- Looking for attempts to use black listed functions.
- Verifying the existence of a properly formed module header as shown in figure 1.
- Warning when modules attempt to use global variables.
- Detecting some memory leaks as shown in figure 2.
- Detecting some dereferences of invalid pointers.

Attempts to refine the memory checks in the list above introduced a new level of complexity to the analysis. Over time these memory checks evolved into the Lighthouse [37] analysis built upon an assumption of exclusive memory ownership in SOS. The analysis performs an instruction level data flow analysis of SOS programs to track all pointers to heap data. Specifications describing how a called function manipulates heap data are used to facilitate a modular analysis that checks each function individually. Pointer aliases are tracked through a combination of separate may and must pointer analysis.

```
mod_op = (sos_module_op_t*) ker_msg_take_data(msg);
if(mod_op == NULL) return -ENOMEM;
if(mod_op->op == MODULE_OP_INSMOD) {
    existing_module = ker_get_module(mod_op->mod_id);
    if(existing_module != NULL) {
        uint8_t ver = sos_read_header_byte(
                existing_module->header,
                offsetof(mod_header_t, version));
            if (ver < mod_op->version) {
                ker_unload_module(existing_module->pid,
                        sos_read_header_byte(
                        existing_module->header,
                        offsetof(mod_header_t, version)));
            } else {
                return SOS_OK;
            }
        }
    ret = fetcher_request(KER_DFT_LOADER_PID,
            mod_op->mod_id,
            mod_op->version,
            entohs(mod_op->size),
            msg->saddr);
    s->pend = mod_op;
    ker_led(LED_RED_TOGGLE);
    return SOS_OK;
}
return SOS_OK;
```

Figure 3: Sample block of code from the SOS module loader containing multiple memory leaks

Table 1: Warnings in SOS user modules

Verified memory leaks identified by analysis	8
False memory leaks identified by analysis	8
Verified dangling pointers identified by analysis	0
False dangling pointers identified by analysis	9

An example bug found in SOS by Lighthouse is shown in figure 3. This code was part of the SOS module loader and fails to free the data allocated into `mod_op` by the call to `ker_msg_take_data` on all paths through the code. The Lighthouse analysis returned the error:

```
Error: Expression mod_op is not stored after
        instruction #line 125
mod_op = (sos_module_op_t *)
        ker_msg_take_data((unsigned char)18, msg);
```

Table 1 lists the results of running the Lighthouse analysis on all versions of all modules in SOS. Table 2 lists the results of running the Lighthouse analysis on the SOS kernel. Most false positives in these analysis resulted from limitations of the must and may alias analysis.

This evaluation reveals that most SOS programs inherently follow a model of exclusive memory ownership and that the implemented analysis is effective at finding memory misuse in SOS.

Table 2: Warnings in the SOS kernel

Verified memory leaks identified by analysis	2
False memory leaks identified by analysis	22
Verified dangling pointers identified by analysis	0
False dangling pointers identified by analysis	11

```
void* malloc_wrapper(int size) {
    void *buffer;
    buffer = malloc(size);
    return buffer;
}
```

Figure 4: Function wrapping `malloc`

Additional details on this version of the Lighthouse checker can be found in the technical report at `http://nesl.ee.ucla.edu/document/show/213`.

4.3 Understanding the Basic Data Flow

The Lighthouse framework uses traditional data flow and static analysis techniques. The analysis begins with a specification describing the required heap state before a function call and the modified heap state resulting from a function call. These specifications allow functions to be analyzed in isolation.

For example the function `malloc_wrapper` listed in figure 4 is verified using the specification in figure 5. This specification states that:

1. There are no global stores referring to heap data
2. `malloc_wrapper` does not depend on any state to be called
3. `malloc_wrapper` returns a pointer to dynamically allocated memory

```
stores {};
malloc.pre {}
malloc.post { $return.full(); }
free.pre { $1.full(); }
free.post { $1.mpty(); }
malloc_wrapper.pre {}
malloc_wrapper.post { $return.full(); }
```

Figure 5: A sample Lighthouse specification

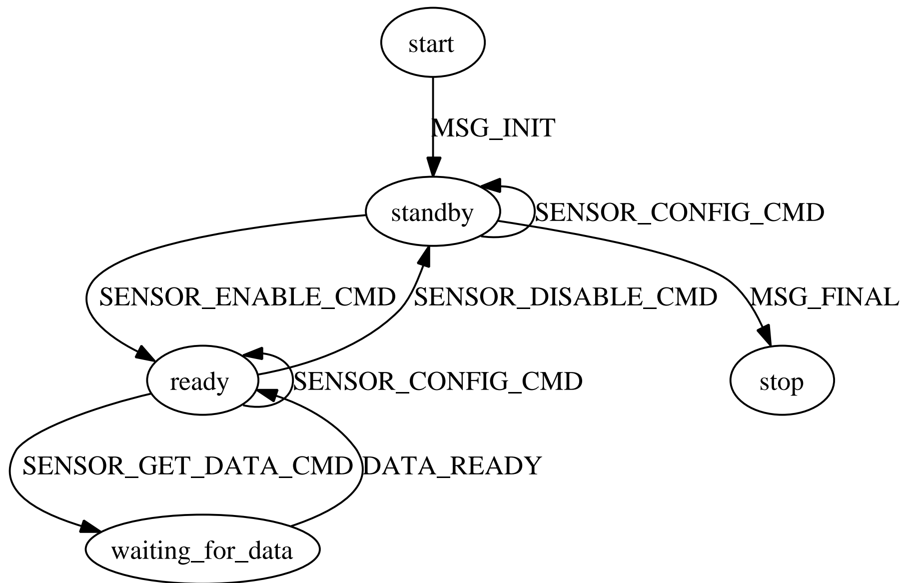

Figure 6: Sensor driver protocol expected by SOS with abstract node states and SOS messages labeling edge transitions

The specification also explicitly lists other functions that manipulate heap memory. Analysis of the `malloc_wrapper` function proceeds by:

1. Initializing the state analysis state to satisfy the pre-conditions for the function being verified. In the example of `malloc_wrapper` this is the empty state.

2. A data flow analysis then examines the body of the function. Upon encountering the call to `malloc` it:

   (a) Verifies that the current state of this function satisfies the preconditions for the called function. This is trivially true for the call to `malloc` which has no preconditions.

   (b) Updates the state to reflect the post condition generated by the called function. For `malloc` this updates the state to reflect that the returned value is `full`, indicating that it points to heap data.

3. At the end of the data flow, each return point is checked to ensure that it adheres to the post conditions in the specification for the function being checked. In this example the data returned must be `full`, which is true.

This approach does have limits. Figure 6 illustrates the protocol that sensor drivers should adhere to. Note that a sensor driver should not be removed from the system while it has an outstanding

```
 1 digraph sensorDriver {
 2    // 185
 3    start -> standby [label="MSG_INIT"];
 4    // 165
 5    standby -> standby [label="SENSOR_CONFIG_CMD"];
 6    // 157
 7    standby -> ready [label="SENSOR_ENABLE_CMD"];
   ...
18 }
```

Figure 7: Ordering specifications are currently written as `dot` graphs

ADC request. This higher level event ordering information is not captured by the pre- and post-condition specifications. Applications not respecting this ordering can not be revealed using the static verification framework described in this section. Recent work on Lighthouse integrates higher level event ordering information into the analysis to alleviate this problem.

5 Future Work

My work on Lighthouse over the past two years has created an analysis framework that integrates traditional static analysis. Application of this framework has focused on dynamic memory usage within sensor networks. My dissertation work expands upon this foundation by exploring the expressibility of the Lighthouse framework.

5.1 Joining Static Analysis and State Machine Specifications

Section 4.3 notes recent work to integrate higher level event ordering into the analysis framework. This is active work that will be completed over the next two months. Specifications of event ordering provides higher level knowledge that the analysis can leverage to make stronger assumptions about the code. This information comes at the expense of needing to augment the verified system with dynamic checks to ensure conformance to the specification.

An ordering specification is described as a state machine with edges corresponding to the firing of events. Figure 7 provides an excerpt from the specification of the sensor driver illustrated in figure 6. A prototype addition to Lighthouse augments user code with a function encoding the state machine, as shown in figure 8, and run time checks, as shown in figure 9. For example if the sensor driver is currently in the `standby` state and receives a `SENSOR_ENABLE_CMD` then:

```
10 bool valid_transition(enum State old, enum State new) {
   ...
30     if (old == dfs_standby) {
31         if (new == dfs_stop) {
32             return true;
33         } else if (new == dfs_ready) {
34             return true;
35         } else if (new == dfs_standby) {
36             return true;
37         } else {
38             return false;
39         }
40     }
   ...
57
58     return false;
59 }
```

Figure 8: Function generated from state machine specification checks for valid state transitions

```
139 static int8_t accel_control(func_cb_ptr cb, uint8_t cmd, void* data) {
140
141     uint8_t ctx = *(uint8_t*)data;
142
143     switch (cmd) {
    ...
157         case SENSOR_ENABLE_CMD:
158             if (valid_transition(s->dfs_state, dfs_ready) != true)
159             { exit(-1); }
160             s->dfs_state = dfs_ready;
161
162             accel_on();
163             break;
    ...
190     }
191     return SOS_OK;
192 }
```

Figure 9: Code accompanying state machine specification is augmented with checks to verify valid state transitions

1. The `SENSOR_ENABLE_CMD` handler on line 157 of figure 9 is executed.

2. This calls the `valid_transition` function with the `standby` state stored in `s->dfs_state` and the destination state as `dfs_ready`.

3. The `valid_transition` function recognizes this as a valid transition on line 33 of figure 8 and execution continues as normal.

In contrast, if in the example above the prior state stored in `s->dfs_state` had been `waiting_for_data`, the call to `valid_transition` would fail to recognize a valid transition, and result in an error.

By having this higher order specification, the underlying static analysis can make stronger assumptions about the program. For example, the dynamic memory checker implemented by Lighthouse need not examine the effects on heap state resulting from a transition from `waiting_for_data` to `stop`, since it is not a valid transition. In this example the addition of a higher order specification can reduce false positives and streamline the execution of the dynamic memory checker.

5.2 New Directions for Lighthouse

The core of the Lighthouse analysis framework is the combination of:

1. Standard static analysis techniques to examine state local to an event.

2. State machine specifications to reason about interactions between events.

3. A union between these two items to create a significantly stronger analysis.

This combination is critical for accurate analysis in the inherently event driven sensor network domain. My dissertation will demonstrate how this analysis framework applies to other challenges in the sensor networking domain by pursuing one or more of the following ideas.

5.2.1 Run Time Memory Usage

Run time memory usage is a potential property for my dissertation to explore. The event based nature of the domain limits the scope of traditional analysis that have difficulties reasoning about interacting asynchronous events. My work would join current stack and run time heap analysis with higher-order information concerning the entire system. This type of analysis is particularly

Table 3: Memory available on different mote platforms

Platform	Microcontroller	RAM	Program Flash	EEPROM	External Flash
Mica2 / MicaZ	ATmega128L 8-bit RISC	4 KB	128 KB	4 KB	512 KB
TelosB	MSP430 16-bit RISK	10 KB	48 KB	16 KB	1024 Kb
Imote2	PXA271 32-bit ARM	256 KB	32 MB	(Flash) 32 MB	

important to sensor networks where limited mote resources, such as those described in table 3, make memory overflows a very real risk.

For systems such as TinyOS that statically allocated program data and queue depths, this analysis provides feedback that developers can use to optimize the size of static data structures. For threaded abstractions emerging in the domain, this work can provide tighter constraints on thread stack requirements. For systems such as SOS that use dynamically allocated data structures, this analysis may be able to accurately approximate worst case memory usage and provide insight into where memory is being used in the system. Systems supporting dynamic loading or updating of applications can use this analysis to help support admission policies on sensor motes.

This analysis would consist of multiple pieces interacting to estimate memory usage. Traditional static stack and heap analysis would be refined to use additional constraints made available through the state machine specification of the program. For example, information from the state machine can help prune paths that a traditional static analysis must examine. The analysis would then rely on the state machine specification to explore the memory usage resulting from different valid interleaving of events in the system. Additional higher level knowledge or load time analysis could be used to reason about how multiple independent applications affect run time memory usage.

5.2.2 Delay and Timing Analysis

Timing is a property well suited to analysis by Lighthouse. While sensor network operating systems are typically not real time systems, some I/O operations are sensitive to jitter or delays in execution. Examples range from signal encoding for infrared and ultrasonic transmitters used in localization, to short sampling bursts that require high accuracy between samples. Two techniques currently used to handle this problem are the placement of time critical code in interrupt handlers and the manual splitting of long running tasks into subtasks. Both approaches have limitations. Shifting critical tasks to interrupts increases the possibility of a long running interrupt blocking another important interrupt. Splitting of an application into smaller subsections can be error prone, even with the help

of abstractions such as protothreads [11].

The Lighthouse analysis framework could be used to analyze delays introduced by various applications and to provide important feedback to system developers. Standard static analysis can be used to approximate the execution times of basic blocks and, with the addition of a data flow, entire functions executing in isolation. Higher level specification information can examine the execution of functions in more complex environments. This includes examining delays resulting from an interrupt interrupting a function and reasoning about the time required to handle a flow of events generated from a single ancestor. This delay and timing analysis helps developers overcome potential bottlenecks in their systems.

5.2.3 Power Consumption

Wireless sensor networks require careful power management to optimize the use of limited power supplies. Current power analysis for sensor network applications, such as PowerTOSSIM [38], combine a power specification of mote hardware with execution traces gathered from simulation. Analysis can provide an alternative approach to solving this same problem. A basic static analysis uses a power specification to generate power consumption summaries for functions. Higher level specifications can then be used to generate a power profile for a proposed execution of the system. The high level specification, used by Lighthouse, facilitates fine tuning by domain experts who can readily see how adjusting run time parameters affect total power usage.

5.2.4 Staged Analysis

Research into staged analysis provides a means to further evolve the Lighthouse framework. The framework described thus far assumes that a higher level specification refines the static analysis, while the resulting run time overhead is an unwelcome but necessary burden. A staged analysis integrates analysis occurring at compile time, load time, and run time into a single framework to allow extensive sharing of information between analysis layers.

This work would explore how established analysis can help and be helped by analysis in other stages. One can imagine combining techniques like software-based fault isolation [36] with more extensive static analysis at compile time to reduce the number of run time checks. Further, properties that fail to pass a static compile time analysis my be reanalyzed at load time or run time when the system may have additional information available to refine the analysis.

5.2.5 Refined Specifications

Finally, this dissertation may naturally refine the specifications used by Lighthouse. Construction of specifications places a significant additional burden on system designers. Further, new opportunities arise for mistakes to enter the system through incorrect specifications.

Outside of sensor networking, a great deal of work has already been done on designing better specifications. Section 3.2.1 mentions work such as statecharts [22] and hierarchical finite state machines [16]. These frameworks provide formalisms that can easily generate code, creating a better correlation between specification and code while easing developer burden.

Alternate work from within the sensor network community, such as protothreads [11] described in section 3.1.3, provides an alternative avenue of approach. Stylized user code, resulting from new language constructs or library support, facilitates automated extraction of expected event firings from code. While generated specifications may require refinement from a developer, they can initialize the specification process and may help specifications to better reflect the underlying code base.

6 Conclusions

This prospectus describes the Lighthouse analysis framework. Lighthouse grew from observing SOS programmers make simple mistakes in their sensor network programs, but soon evolved to handle the challenges of analyzing software in an event driven domain. Initial versions of Lighthouse have been used to help find incorrect use of dynamic memory within both kernel and user code for the SOS operating system.

Over the next two months the Lighthouse analysis framework will finish combining static analysis with state machine verification to create more precise views of a software system. This combination is critical in the event based sensor network domain to prevent the analysis from assuming that any event can fire at any time. The next step will be to apply Lighthouse to new problems. This prospectus proposes examining one or more of the described problems in the sensor network domain to push the limits of Lighthouse.

References

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA*, pages 311–330. ACM Press, 2002.

[2] T. Ball and S. K. Rajamani. SLIC: A specification language for interface checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, January 2002.

[3] G. Berry. The foundations of esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[4] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static data flow. In *Transactions on Signal Processing*, pages 397–408. IEEE Press, 1996.

[5] A. Boulis, C.-C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *MobiSys*, pages 187–200. ACM Press, 2003.

[6] A. Cerpa, N. Busek, and D. Estrin. Scale: A tool for simple connectivity assessment in lossy environments, 2003.

[7] B. Chelf, D. Engler, and S. Hallem. How to write system-specific, static checkers in metal. In *PASTE '02: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 51–60. ACM Press, 2002.

[8] K. Chintalapudi and R. Govindan. Localized edge detection in wireless sensor networks. In *SNPA*, pages 1–11. IEEE Press, 2003.

[9] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *SIGPLAN*, pages 59–69, 2001.

[10] D. Dhurjati, S. Kowshik, and V. Adve. Enforcing Alias Analysis for Weakly Typed Languages. Technical Report #UIUCDCS-R-2005-2657, CS Dept., U. of Illinois, Nov 2005.

[11] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *SenSys*. ACM Press, 2006.

[12] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges: scalable coordination in sensor networks. In *MobiCom*, pages 263–270. ACM Press, 1999.

[13] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity OS. In *EuroSys*. ACM SIGOPS, ACM, 2006.

[14] 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 *PLDI*, 2003.

[15] M. Geilen and T. Basten. Requirements on the execution of kahn process networks. In *ESOP*, 2003.

[16] A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. In *TCAD*. IEEE Press, 1999.

[17] O. Gnawali, K.-Y. Jang, J. Paek, M. Vieira, R. Govindan, B. Greenstein, A. Joki, D. Estrin, and E. Kohler. The tenet architecture for tiered sensor networks. In *SenSys*, pages 153–166. ACM Press, 2006.

[18] L. Gu and J. A. Stankovic. tkernel: Providing reliable os support for wireless sensor networks. In *SenSys*. ACM Press, 2006.

[19] N. Halbwachs and L. Mandel. Simulation and verification of asynchronous systems by means of a synchronous model. In *ACSD*, pages 3–14. IEEE Computer Society, 2006.

[20] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *The Proceedings of the 2002 PLDI*. PLDI, ACM, 2002.

[21] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *MobiSys*, pages 163–176. ACM Press, 2005.

[22] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274, June 1987.

[23] S. O. Howard and J. O. Hallstrom. Specifying and enforcing resource utilization constraints. In *EmNets*. IEEE Press, 2006.

[24] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *7th USENIX OSDI*. OSDI, USENIX, 2006.

[25] E. A. Lee. The problem with threads. Technical Report UCB/EECS-2006-1, EECS, UC Berkeley, January 10 2006.

[26] E. A. Lee and D. G. Messerchmitt. Synchronous data flow. In *Proceedings of the IEEE*, pages 1235–1245. IEEE Press, 1987.

[27] E. A. Lee and S. Neuendorffer. Concurrent models of computation for embedded software. In *IEE Proceedings, Computers and Digital Techniques*, pages 239–250. IEE, 2005.

[28] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *ASPLOS*, Oct. 2002.

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

[30] W. P. McCartney and N. Sridhar. Abstrations for safe concurrent programming in networked embedded systems. In *SenSys*. ACM Press, 2006.

[31] M. Melkonian. Get by without an rtos. *Embedded Systems Programming*, September 2000.

[32] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *Computational Complexity*, pages 213–228, 2002.

[33] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[34] J. Regehr. Thread verification vs. interrupt verification. In *TV*, 2006.

[35] J. Regehr, N. Cooprider, W. Archer, and E. Eide. Efficient type and memory safety for tiny embedded systems. In *PLOS*, page 6. ACM Press, 2006.

[36] R. Rengaswamy, E. Kohler, and M. B. Srivastava. Software based memory protection in sensor nodes. In *EmNets*. IEEE Press, 2006.

[37] R. S. Shea, S. Markstrum, T. Millstein, R. Majumdar, and M. B. Srivastava. Static checking for dynamic resource management in sensor network systems. Technical Report TR-UCLA-NESL-200611-02, UCLA, Novemeber 2006.

[38] V. Shnayder, M. Hempstead, B. rong Chen, G. W. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *SenSys*, pages 188–200, 2004.

[39] B. L. Titzer. Virgil: Objects on the head of a pin. In *OOPSLA*, 2006.

[40] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *SenSys*, pages 14–27. ACM Press, 2003.

[41] J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *SenSys*, pages 1–13. ACM Press, 2003.