2008 International Conference on Information Processing in Sensor Networks

Deriving State Machines from TinyOS Programs using Symbolic Execution

Nupur Kothari*
University of Southern California
nkothari@usc.edu

Todd Millstein
University of California, Los Angeles
todd@cs.ucla.edu

Ramesh Govindan
University of Southern California
ramesh@usc.edu

Abstract

The most common programming languages and platforms for sensor networks foster a low-level programming style. This design provides fine-grained control over the underlying sensor devices, which is critical given their severe resource constraints. However, this design also makes programs difficult to understand, maintain, and debug.

In this paper, we describe an approach to automatically recover the high-level system logic from such low-level programs, along with an instantiation of the approach for nesC programs running on top of the TinyOS operating system. We adapt the technique of symbolic execution *from the program analysis community to handle the event-driven nature of TinyOS, providing a generic component for approximating the behavior of a sensor network application or system component. We then employ a form of* predicate abstraction *on the resulting information to automatically produce a finite state machine representation of the component. We have used our tool, called* FSMGen*, to automatically produce compact and fairly accurate state machines for several TinyOS applications and protocols. We illustrate how this high-level program representation can be used to aid programmer understanding, error detection, and program validation.*

1. Introduction

Understanding the correctness of sensor network applications is difficult, since programmers often have to manage devices and resources at a relatively low-level and be aware of memory, processing and bandwidth constraints. At the same time, these applications are often required to run unattended for long periods of time in harsh environments. Ensuring the reliability of sensor network applications is thus an important problem.

*This author was supported by the USC Annenberg Graduate Fellowship

The sensor network community has responded to this problem in three ways. The first has been to propose high-level programming techniques that can simplify the application programmer's task. These include virtual machines [22], macroprogramming approaches [20, 26], and role-based or state-based programming languages [11, 17]. The second has been to develop run-time monitoring (e.g., Sympathy [27]) and debugging tools (e.g., Nucleus [30], Clairvoyant [32]) that can simplify the process of discovering program errors. The third has been to develop compile-time program analysis tools [28, 9, 8] for catching program errors before execution.

If history is any guide, none of these approaches is likely to be a panacea in and of itself. Today programmers in other domains use a variety of tools ranging from compiler analysis to profilers and debuggers, even though much work has gone into raising the level of abstraction from assembly code to visual programming. Similarly, we believe that for sensor networks in the future, it will be useful to have an arsenal of tools to catch or avoid program errors, and our work is an attempt to add to the existing arsenal.

In this paper, we focus on a program analysis tool for TinyOS software written in nesC. TinyOS is the most dominant application-development platform today and is likely to continue to be so in the near future. In the longer term, even if applications were to be written in a higher-level language, there would still likely be a large installed base of TinyOS software that implements the protocols and subsystems executing on sensor nodes.

Generally speaking, our goal is to infer a user-readable high-level representation of any component of a TinyOS program. Such a high-level representation accurately captures system logic while abstracting away platform-specific details. This goal is motivated by the following observation: when programmers write code, there is often a disparity between programmer intent and the functionality embedded in the resulting code. Such a disparity can arise, for example, when the programmer assumes a certain contract of an interface where none exists, resulting in a (possibly latent) program error. TinyOS's event-driven programming

978-0-7695-3157-1/08 $25.00 © 2008 IEEE
DOI 10.1109/IPSN.2008.62

IEEE computer society

model [21] can exacerbate this problem, since it makes it hard for the programmer to understand the exact sequencing of operations. Our claim is that, when programmers are presented with a high-level representation of TinyOS components they have written, they can much more easily detect such discrepancies.

By a TinyOS *component* we refer to a logical component which may consist of a single TinyOS module, e.g. the Surge module for the Surge application, or of a number of cooperating TinyOS modules, such as the RfmToInt and IntToLeds modules, which cooperate together in the RfmToLeds application. A component can implement application logic, like the above two examples, or a system function, like a routing (MultiHopLQI) or a time synchronization (FTSP).

Inferring a high-level representation from arbitrary code is a significant challenge, but we can leverage current practice in developing TinyOS code: anecdotal evidence suggests that many programmers often design TinyOS software using finite-state machines (FSMs). Although the nesC programming language provides no explicit support for state machines, programmers track event execution by explicitly maintaining state information (as we show in Section 2). Thus, our specific goal in this paper is to *infer compact, user-readable FSMs corresponding to TinyOS applications and system components.* Our paper makes the following contributions towards this goal.

Novel Program Analysis. The programming languages community has developed many general-purpose techniques for program analysis. Two of these techniques are *symbolic execution* and *predicate abstraction*. The former precisely simulates a program's execution and maintains symbolic information about the program, while the latter maps this symbolic information into predicates that define distinct program states. Our contribution is two-fold. First, we have adapted symbolic execution to TinyOS's event-driven programming model. This entailed approximating the flow of control of an application, which is complicated due to the two-level scheduling structure of TinyOS with events and tasks and due to split-phase operations. To address this issue, we employ a simple model of event-driven execution that is precise enough to capture important program behaviors yet abstract enough to be user-understandable. Second, we have used predicate abstraction to generate compact, user-readable state machines; prior work [1, 13] has focused on generating state machine representations as an internal step within a larger verification effort.

Tool Design, Implementation, and Evaluation. We have designed a tool called *FSMGen* that contains a symbolic execution framework for TinyOS programs and employs predicate abstraction, together with an aggressive state-machine minimization technique, to infer user-readable FSMs for any component of a TinyOS program (Sections 3 and 4). *FSMGen* is well-suited to infer the higher-level system logic and functionality of components such as, for example, how an incoming message is dealt with in a routing protocol. However, since we use a relatively coarse approximation of the TinyOS event-based execution model, it cannot precisely capture the functionality of low-level interrupt-driven code, like that of the timer component in TinyOS, or the radio component. We have applied *FSMGen* to a variety of TinyOS programs, generating FSMs for components ranging from simple applications like RfmToLEDs, Surge, and TestNetwork to a routing protocol of moderate complexity (MultiHopLQI) and a fairly complex time synchronization protocol (FTSP). We qualitatively discuss the performance of the tool and show how the inferred FSMs reveal surprising (and, we believe, previously unknown) aspects of some of these components (Section 5).

2. Overview

In this section we provide an overview of our approach to inferring finite state machines for TinyOS components. We begin by discussing the suitability of FSMs as high level program representations of TinyOS components, provide a definition of an FSM for TinyOS components, and highlight our contributions using a simple example.

2.1. FSMs as abstractions of TinyOS components

The event-driven programming model enforced by TinyOS is qualitatively different from the thread-based programming models that most programmers are familiar with. To understand and reason about their TinyOS applications or system components, written in nesC, anecdotal evidence suggests that programmers often use a finite-state machine (FSM) based design approach. In such an approach, programmers design applications/protocols as finite state machines and then embed these within nesC code. A TinyOS program may thus consist of multiple FSMs, interacting with one another, each representing the functionality of a single logical component. The programmer maintains state information explicitly as program variables. On receipt of an event, an event handler performs the appropriate action depending on the current state and transitions to a new state by updating variables. Indeed, the developers of TinyOS recognized the relationship between FSMs and the event-driven programming model, as this excerpt from their paper [15] shows:

> ... in that the requirements of an FSM based design maps well onto our event/command structure.

```
1   event result_t Timer.fired() {
2     if (initTimer) {
3       initTimer = FALSE;
4       return call Timer.start(
5           TIMER_REPEAT, timer_rate);
6     }
7     timer_ticks++;
8     if (timer_ticks %
9        TIMER_GETADC_COUNT == 0) {
10      call ADC.getData();
11    return SUCCESS;
12  }
13  task void SendData() {
14    if (..) {
15      if ((call Send.send(..) != SUCCESS)
16              atomic gfSendBusy = FALSE;
17    }
18  }
19  event result_t ADC.dataReady
20        (uint16_t data) {
21    atomic {
22      if (!gfSendBusy) {
23        gfSendBusy = TRUE;
24        gSensorData = data;
25        post SendData();
26      }
27    }
28    return SUCCESS;
29  }
30  event result_t Send.sendDone(...) {
31    atomic gfSendBusy = FALSE;
32    return SUCCESS;
33  }
```

Figure 1. FSM embedded within Surge code

This relationship to FSMs is evident in TinyOS application code. Consider, for example, the snippet of code in Figure 1, taken from the Surge application in TinyOS. Surge periodically (on the `Timer.fired` event) tries to get readings from a sensor. On receipt of the data from the sensor (on the `ADC.dataReady` event), Surge routes it back to the base station (using the `Send.send` function). The variables `initTimer` and `gfSendBusy` represent the explicit state of the Surge application, as maintained by the programmer. In the event handler for the `ADC.dataReady` event, if `gfSendBusy` is `TRUE`, that implies that a packet is currently being sent, and hence the programmer does not try to send another packet. However, if `gfSendBusy` is `FALSE`, the programmer sets `gfSendBusy` to `TRUE` and uses the `Send.send` command to send the data back to the base station. When the event `Send.sendDone` is triggered, i.e. the data has been sent, the programmer resets `gfSendBusy` to `FALSE`.

In addition to this state explicitly maintained by the programmer, an application's high-level FSM is dependent on the set of external events that can be signalled at each point. For example, in Surge, before the command `Send.send` is called, the program is in a state where the external event `Send.sendDone` cannot occur, since `Send.send` and `Send.sendDone` form a split-phase operation. When the command `Send.send` is called, the program moves into a new state, where the set of possible external events includes `Send.sendDone`. Here, by external events, we refer to events which are triggered from outside the component whose functionality is being examined. These may be events which form part of a split-phase operation, like the `Send.sendDone` event in the above example and hence are triggered indirectly by a command in the component, or events that are not triggered (directly or indirectly) by any action performed within the component, e.g. an event indicating the reception of a packet.

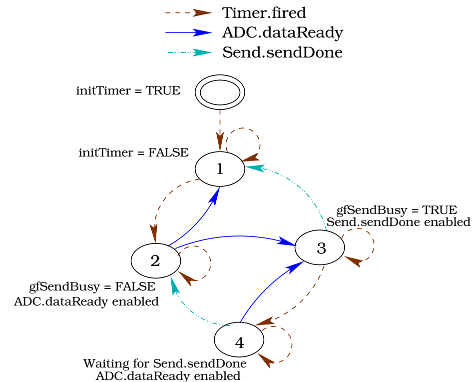

Figure 2. FSM derived manually for Surge

Thus, the state of execution of TinyOS components is a combination of explicitly maintained program variables and the set of external events possible at that point in execution. This suggests the following hypothesis, which we validate in this paper: *armed with limited domain-specific information about external events, it is possible to infer a user-readable finite-state machine corresponding to a given TinyOS application or system component written in nesC.*

2.2. Deriving FSMs from TinyOS programs

In this section, we present an overview of our technique for inferring FSMs for various components from TinyOS programs. Figure 2 depicts an FSM for the Surge application. This FSM was manually derived from the Surge application code, by focusing on the Surge component.

Each state is intuitively defined by a combination of the explicitly maintained state information of the component and the set of enabled external events at that point. For example, the initial state of the FSM (denoted by the double circle) represents the situation when `initTimer` is set to `TRUE`, and the `Timer.fired` event is enabled. When the event handler for `Timer.fired` is invoked with the program in state 0, `initTimer` is set to `FALSE` and we transition to state 1.

Each edge is labelled with an event, along with an optional predicate about the associated event handler's execution. For example, the transition from state 1 to 2 only

occurs when the `Timer.fired` event occurs and the call to `ADC.getData()` within the associated event handler returns `SUCCESS`. If the `Timer.fired` event occurs and the call to `ADC.getData()` returns `FAIL`, the program remains in state 1. In Figure 2, for the sake of simplicity, we have not labelled the edges with predicates, since the states they connect provide sufficient intuition.

One approach to inferring FSMs in the literature [2] is to dynamically monitor a component to capture the order of events and the associated values of program variables when these events occur. This information is then fed to a machine learning algorithm to infer the states and state transitions. However, a dynamic approach is inherently incomplete, since an application can have an infinite number of execution traces. Therefore, the results can easily represent the particular runs of the application that occur during monitoring but fail to capture other program behaviors.

We have pursued an alternate approach based on *static analysis* of an application. Static analysis has the advantage that it can conservatively consider all possible program executions, including corner cases that could easily be missed in a dynamic approach. Inferring FSMs statically requires two key challenges to be addressed:

- How can we obtain precise information about a component's execution without running the application? This challenge is exacerbated by the TinyOS execution model, with its asynchronous execution of *tasks* and the possibility of hardware interrupts at any point.
- How can we automatically identify the relevant state information of a component whose FSM we intend to extract, and how can we represent the component's behavior in terms of this state information once it is identified?

We address these challenges by adapting and extending two techniques from the programming languages literature.

First, we precisely track the behavior of a component whose FSM we are interested in, via *symbolic execution* of its TinyOS program. This static analysis technique employs a constraint solver to precisely simulate the program's execution, maintaining symbolic information about the values of program variables. Unlike a dynamic analysis, symbolic execution conservatively considers the behavior of all possible program executions while pruning many infeasible paths from consideration. We have designed and implemented a generic framework for symbolic execution of TinyOS programs. The next section describes this framework in detail.

Second, we use a technique called *predicate abstraction* to map the program information as tracked by symbolic execution into a finite set of predicates that capture the important state information for the component of interest. The predicates are automatically derived from the branch conditions in the system logic of the component. We discuss this technique in detail in Section 4. In addition, we use an adapted version of a well-known FSM minimization technique (the Myhill-Nerode algorithm [25]) to merge "similar" states, resulting (as we show later) in user-readable finite state machines.

3. Symbolic Execution for nesC/TinyOS

Symbolic execution is a program analysis technique that statically approximates the behavior of a program. Informally, the technique involves simulating the execution of a program without actually running it, maintaining at each point information about the value of each variable. Because of its generality, symbolic execution has a wide variety of applications for reasoning about programs. We show in the next section how to use the results of our symbolic execution to automatically derive finite-state machines for user-specified components from TinyOS programs.

Our symbolic execution framework is built as an interprocedural analysis in the CIL [24] front end for C. Our framework takes as input, the C file generated as part of the building process for a TinyOS application using the nesC compiler. The framework simulates execution of this program starting from `main`. We assume the user designates certain modules (and hence the functions in those modules) as *interesting*, meaning that they are part of the component being analyzed. As we discuss below, uninteresting functions are not traversed during the symbolic execution, but are instead treated conservatively.

Symbolic execution is necessarily approximate. For example, it is not possible in general to know the exact value of each variable at every program point. Instead the symbolic execution maintains a *symbolic store*, which maps variables to *symbolic values*, which are values that can refer to *symbolic constants*, denoted c_i. For example, at some point we may know that x has the value c_x and y has the value $c_x + 5$. Further, it is not possible in general to know which path a program will take at a branch point (e.g., a conditional or loop), so the symbolic execution framework must simulate multiple paths. At each program point, the current path is represented by a set of predicates (which can include symbolic constants) that are assumed to be true. The predicates are simply the branch conditions that led to this point on the path.

While symbolic execution has been implemented for C [31, 10], to our knowledge ours is the first symbolic execution framework to handle the unique features of nesC and TinyOS. We first describe the basic technique of symbolic execution, which is relatively standard. Next we describe the ways in which we extended symbolic execution to handle nesC- and TinyOS-specific features. Finally we describe the constraint solver that we use as part of the symbolic execution, in order to prune infeasible execution paths.

3.1. Basic Symbolic Execution

Let a *symbolic state* be a pair of a symbolic store and a set of predicates representing the current path. The result of symbolic execution is the determination of a set of symbolic states for each point in the program, representing the possible runtime states that could arise during execution at that point. In the rest of this subsection we discuss how symbolic execution handles standard C language constructs.

Assignments To symbolically execute an assignment $x := e$, we evaluate e in the current symbolic store to some symbolic value v and update the symbolic store so that x maps to v. If the left-hand side is an array update $a[e_a]$, then the framework tries to evaluate e_a to a numeric constant in the current symbolic store. If it is able to do so, then that array element is updated appropriately. Otherwise, all information about the entire array is conservatively removed from the symbolic state. Assignments through pointers are handled similarly.

Conditionals The framework invokes a constraint solver to determine the value of the conditional's guard expression e in the current symbolic state. If the solver determines that the guard is true, then symbolic execution proceeds on the "then" branch, and on the "else" branch if the solver determines that the guard is false. If the solver cannot determine e's value, then the current symbolic execution bifurcates. The framework adds e to the set of predicates assumed to be true and continues traversal of the "then" branch. Separately, the framework instead adds $!e$ to the set of predicates assumed to be true and continues traversal of the "else" branch. We use a work queue to keep track of pending paths to be traversed.

Function calls The function call's actual argument expressions are evaluated to symbolic values in the current symbolic store. If the function being called is part of an *interesting* module, then the symbolic store is updated with a mapping from the function's formals to the symbolic values of the actuals, and traversal proceeds inside the function body. When the traversal eventually hits a `return` statement (or the end of the function), control transfers back to the caller, and the returned value (if any) is handled like an assignment statement.

If the function is not designated as interesting, then we do not traverse the function body. Instead we use a precomputed summary of the function body (which we compute before beginning the traversal), which indicates variables in the caller's scope that might be invalidated by the call, in order to conservatively "kill" facts in the current symbolic state. Our framework currently does not deal with recursive functions.

Loops As with conditionals, the framework invokes the constraint solver to determine the value of the loop's termination condition. If the value is true, then traversal continues after the loop. If the value is false, then traversal continues inside the loop (and returns to the top of the loop upon reaching the end). In this way, we precisely simulate bounded loops (e.g., simple `for` loops), which we have found to be the common case in TinyOS applications. If the solver is unable to precisely evaluate the termination condition, then we simply traverse the loop exactly once. This is done in order to identify nesC tasks and events that are triggered within the loop (see the next subsection). This simple approach loses information about potentially signaled events, but it has not been a large problem in practice. To continue symbolic execution conservatively after the loop, we invalidate all information in the resulting symbolic state about variables that are potentially modified within the loop body.

3.2. Handling features of nesC and TinyOS

The nesC language and TinyOS platform pose several challenges for performing accurate symbolic execution. We discuss the key features of these tools and how our symbolic execution framework handles them.

Tasks TinyOS *tasks* are a form of asynchronous function. Posting a task pushes a pointer to the task into a *task queue* maintained by the TinyOS runtime. Tasks from this queue are dequeued and executed (in FIFO order) whenever there is nothing else running.

Our symbolic execution framework mirrors this approach. We augment each symbolic state with a task queue. When we encounter the posting of a task during traversal, we simply treat it as a no-op and proceed to the next statement. However, we add the task to the queue. Once this path of execution has been completely simulated, we pop tasks off the queue in FIFO order and simulate the execution of each in succession. Of course, the simulation of a task may in turn cause more tasks to be added to the queue recursively. Simulation continues until the task queue is empty.

Events Our symbolic execution framework must track the events that can fire at any point during the program, in order to properly simulate program execution. There are two main flavors of events in TinyOS, and we handle each in a different manner.

First, many events are simply triggered by a direct call from within the program (often from within a task). These events are treated as ordinary function calls, with the traversal continuing inside the corresponding event handler.

Second, at each program point, our symbolic execution framework maintains a set of possible external events that

may fire. It would be too unwieldy to consider the possibility of these events being handled at each program point. Instead, our framework assumes that such events will only be processed once a prior event handler and all posted tasks have completed their execution and the application is "waiting" for a new event. At that point, our symbolic execution framework explores all possible orders in which the enabled external events may be processed. While this approach can miss potential execution paths, if the programmer ensures that no interrupt handler unwittingly modifies any variables used within a task that it can interrupt, the resulting symbolic state after some missing execution path will be identical to that of some execution path that our model does consider. Possibly for this reason, we have not noticed the loss of precision in practice.

Maintaining the set of enabled external events requires tracking two kinds of external events, as described in Section 2. External events that are not triggered within the program, but instead can occur at any time, are always considered in the set of enabled events. Apart from these, events forming part of a split-phase operation are considered in the set of enabled external events only if the command triggering them is executed.

We assume that the user provides our framework with the set of split-phase event pairs, in order to complete our knowledge about external events. In our experiments, we only needed to provide at most five such event pairs. When a call to a split-phase operation (e.g., `Send.send`) is encountered during traversal, we traverse the corresponding handler as described above. Upon returning to the caller, we invoke the constraint solver to determine the value of the result. If we determine that the result indicates success, then we add the corresponding external event (e.g., `Send.sendDone`) to the set of enabled external events. If we determine that the result indicates failure, then the external event is not added to the set. If the value of the result cannot be determined, then we simulate both possibilities.

3.3. Constraint Solver

As mentioned above, we use a constraint solver to determine the values of predicates during the traversal, in order to prune infeasible paths. Rather than building our own customized tool, we use an off-the-shelf constraint solver, CVC3 [29]. This tool incorporates decision procedures for a variety of logical theories, including propositional logic, linear arithmetic, bit vectors, arrays, and structures.

To determine the value of a predicate e in a given symbolic state, our framework automatically mirrors the symbolic state as axioms that are provided to CVC3. For example, if the symbolic store maps y to the symbolic value $c_x + 5$, then we declare variables y and c_x in CVC3 along with the axiom $y == c_x + 5$. Similarly, each predicate in the symbolic state's set of assumed predicates is translated into a CVC3 axiom.

Finally, the predicate e is translated to CVC3 and posed as a query. If CVC3 indicates that e is valid in the context of the given axioms, then we know that e has the value true at this point. Otherwise, we pose $!e$ as a query to CVC3. If CVC3 indicates that $!e$ is valid in the context of the given axioms, then we know that e has the value false at this point. Otherwise, we consider the value of e to be unknown.

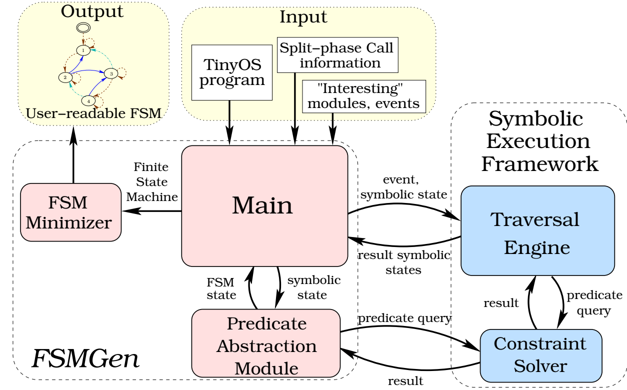

Figure 3. Structure of *FSMGen*

4. Deriving State Machines with *FSMGen*

Figure 3 describes the overall structure of *FSMGen*, our tool for automatically inferring FSMs for TinyOS applications or system components. In addition to the TinyOS program, *FSMGen* requires domain-specific information about commands and events that are split-phase, as mentioned earlier, since this information is not derivable from the code. *FSMGen* also requires the user to annotate modules to be considered *interesting* for the component whose FSM they want to extract, and to list the events they want included in the resulting state machine. *FSMGen* interacts with the symbolic execution framework described in the previous section to obtain symbolic states at various program points of interest. It also reuses that framework's constraint solver to perform *predicate abstraction*, which maps each symbolic state to a state of the resulting FSM. Finally, *FSMGen* employs a minimization procedure to make the FSM compact and user-readable.

We first describe how *FSMGen* derives and uses a finite set of predicates as the basis for each state in the FSM. We then describe the algorithm that *FSMGen* uses to build the FSM, utilizing the symbolic execution framework and predicate abstraction. Finally, we describe *FSMGen*'s algorithm for minimizing the state machine produced by the previous step.

4.1. Predicate Abstraction

As described in Section 2, each state in the FSM should represent a set of predicates about the program state. We use a simple but effective approach to deriving the appropriate predicates to employ.

First, we collect the set of predicates used as guards in conditional expressions within modules declared interesting. Intuitively these predicates are important since they determine the flow of control through the interesting modules, thereby also determining how program state is updated and which events are signaled. Since the states in the FSM are global to the entire application, we remove from this set any predicate that does not refer to a global variable or a formal parameter of a function.

Second, we introduce one additional predicate for each split-phase operation provided by the user, which tracks whether we are in the middle of such an operation (e.g., `send` has been signaled and we are waiting for a `sendDone` event). These predicates effectively track the set of enabled external events at any program point.

Let us denote this set of predicates as $\{e_1,\ldots,e_n\}$. These predicates induce an FSM with 2^n states, one for each possible valuation to the n predicates. *FSMGen* employs our symbolic execution framework to determine the relationships among these states, as described below. A key piece of *FSMGen*'s algorithm is *predicate abstraction*, which maps a symbolic state obtained from symbolic execution to the corresponding FSM state. Given a symbolic state, we employ the constraint solver as described earlier to obtain the valuation of each of the predicates in $\{e_1,\ldots,e_n\}$. Since in general some of these predicates might not be known to be either definitely true or definitely false in the given symbolic state, predicate abstraction in fact maps a symbolic state to a set of FSM states in which the program might be.

4.2. Generating the FSM

To begin, *FSMGen* uses the symbolic execution framework to analyze the `main()` function of the program. Predicate abstraction is applied to each of the returned symbolic states, and the resulting FSM states form the initial states of the FSM. Each FSM state is put on a work queue. Further, for each FSM state, *FSMGen* records the associated symbolic states and the list of enabled events at this point, both of which were obtained from the symbolic execution framework.

After this initialization phase, the main loop of *FSMGen* begins. An FSM state is removed from the work queue, and the symbolic execution framework is asked to simulate each enabled event, starting from each recorded symbolic state. For each such query, the symbolic execution framework returns a list of new symbolic states as well as the new set of enabled events. *FSMGen* employs predicate abstraction on each symbolic state and adds a transition to the FSM from the original FSM state to each resulting FSM state, labelled with the simulated event. The label also includes any new predicates that are part of the symbolic state returned from the symbolic execution framework, which represent the conditions under which the state is reached when that event is invoked. If this transition does not already exist in the FSM, then the new FSM states are added to the work queue. The algorithm continues in this way until the work queue is empty.

We choose to start symbolic execution from the recorded symbolic state rather than the FSM state, for each state in the work queue, in order to have access to the extra information provided in the symbolic state. This extra information allows us to statically prune away transitions which may never be taken at run-time, and hence generates more accurate transitions. However, we only put the resulting FSM state onto the work queue if this FSM transition does not already exist, even if the symbolic state has changed. This choice can cause us to miss possible edges in the FSM. An example of this limitation was observed for the FTSP protocol described in Section 5. We are currently exploring ways to balance this tradeoff between the precision and completeness of our algorithm.

4.3. Minimizing the FSM

Finally, we employ a variant of the Myhill-Nerode FSM minimization algorithm on the FSM resulting from the above algorithm. The basic idea of that algorithm is to identify equivalence classes of FSM states that can be merged without loss of information. The algorithm works by initially assuming that all states belong to one equivalence class. It then looks at each pair of states (s_1,s_2) to see if they can in fact belong to the same equivalence class. In the Myhill-Nerode algorithm, this is the case if they agree on their outgoing edges. For example, if s_1 has an outgoing edge labelled l to state s_3, then s_2 must also have an outgoing edge labelled l to a state in the same equivalence class as s_3. A label in our context is a pair of an event and the associated conditions under which this edge is taken.

Our algorithm proceeds similarly, except that we place one additional requirement on each pair of states: If they have incoming edges from equivalent states labelled with the same event, then the labels must also agree on the associated conditions. Myhill-Nerode does not constrain incoming edges, since this does not affect the language accepted by the FSM. However, in our setting we care not only about the language accepted by the FSM, but also about what predicates hold at each point during program execution (i.e., which state we are in). We have found this new requirement on incoming edges to be a useful heuristic for

minimizing FSMs while retaining important state information. However, it also causes less minimization than would otherwise be performed, so we allow the user to disable it.

5. Results

In this section, we describe our evaluation of *FSMGen*. Our evaluation is qualitative and aims to demonstrate the practicality of *FSMGen*, the compactness and user-readability of the resulting FSMs even for some sophisticated programs, and the utility of *FSMGen* in highlighting interesting and sometimes unexpected features of popular TinyOS applications and protocols. In addition, we discuss various aspects of symbolic execution, predicate abstraction, and minimization that manifest themselves in the generated FSMs.

We used *FSMGen* to infer FSMs for many TinyOS applications and system components. The selected TinyOS programs covered a range of complexity, from simple applications like RfmToLeds, to complex protocols like FTSP [23]. *FSMGen* took at most 15 minutes to analyze all but one program. We discuss this exception later in the section. *None of our inferred FSMs exceeds 16 states.*

RfmToLeds Figure 4 depicts the FSM inferred by *FSMGen* for the RfmToLeds application in TinyOS-1.x. This application listens for packets containing a byte-sized value. When it receives such a packet, the application activates mote LEDs in the appropriate binary pattern. Our FSM captures this functionality accurately. State 0 is the initial state where the program waits to receive a packet. On receiving the packet, depending on the value contained therein, it moves into one of the others states, turning on/off the appropriate LEDs.

Throughout this section, our graphical depictions of the FSMs include state and edge labels that are slightly simplified, using some information from the application code for expository purposes. For example, the labels on each state in Figure 4 indicating the corresponding LED configuration in fact correspond to conditions on edges in the *FSMGen*-generated FSM. Figure 5 zooms in on the two transitions between states 0 and 4, showing the actual edge conditions in the FSM output by *FSMGen*. Here, `value` is a local variable within the event handler for `Receive.receive`, which depends on the packet received. We emphasize that, to a programmer familiar with the actual code, the output of *FSMGen* is highly readable.

This application also illustrates another feature of *FSMGen*. There exists a correct, but less informative, 2-state FSM for this application: in the initial state, the program waits for a packet, and a second state in which it activates LEDs. *FSMGen* can generate this more compact

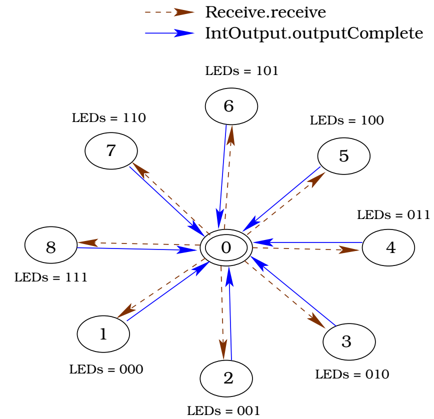

Figure 4. FSM for the RfmToLeds Application

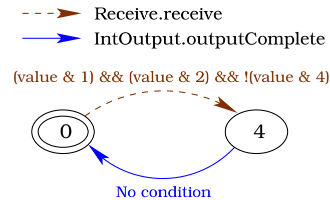

Figure 5. A state transition as generated by *FSMGen* for the RfmToLeds FSM

FSM using the unmodified Myhill-Nerode minimization algorithm described in Section 4.

Surge Figure 6 shows the FSM generated for the Surge example that we had described in Section 2. When we compare this with the manually-produced FSM in Figure 2, we notice that most of the states and transitions in the two FSMs match, but the FSM generated by *FSMGen* has two extra states, 4, and 6. Interestingly, once the program has moved into either states 4 or 6, it stays in one of those states. These two states and the associated edges represent a path of execution that we did not expect to encounter.

To understand this execution path, we examined the application code. The only way for the program to move into state 4 is via the edge from 2 to 4 on the `ADC.dataReady` event. This transition is taken in the task `sendData` when the value returned by the call `Send.getBuffer` is 0. When this happens, the program exits the task without sending any data, but does not reset `gfSendBusy` to `FALSE`, so the program incorrectly assumes that data is being sent, and remains waiting for `sendDone`.

It this a program error? Quite possibly. Surge uses MultihopLQI, which provides the `getBuffer` interface. In the current implementation `getBuffer` never returns 0, so the

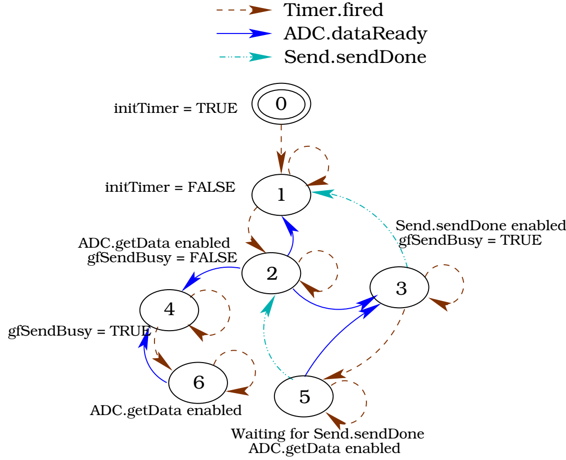

Figure 6. FSM for the Surge Application

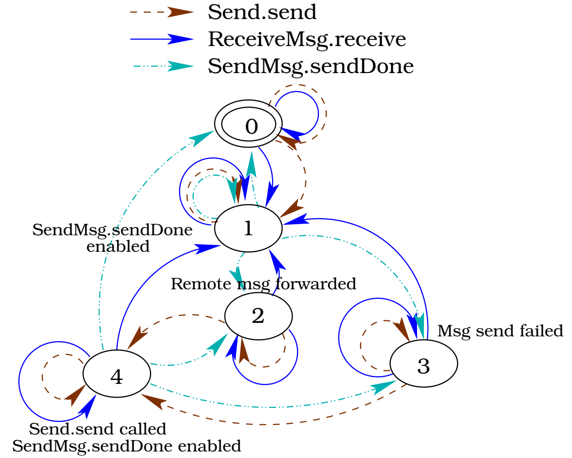

Figure 7. FSM for MultiHopEngine

edge from state 2 to 4 would never be taken. However, the programmer seems to have anticipated the fact that if the underlying implementation were to change, `getBuffer` might return 0, and added a check for the return value in the code. But in forgetting to reset the `gfSendBusy` variable to FALSE, the programmer has introduced an anomaly (at best, and a latent bug, at worst) into Surge, one that was not readily apparent upon manual inspection of the code.

MultiHopEngine MultiHopEngine is the component which acts as a packet forwarding engine for the MultiHopLQI and MintRouting routing protocol implementations in TinyOS-1.x. This component provides a `Send` interface for the programmer to send packets to it. It then forwards these packets to the `SendMsg` interface, which sends them over the network to the next hop in the routing tree. Also, it forwards packets received from the network over the `ReceiveMsg` interface to the `SendMsg` interface. Unlike our prior examples, MultiHopEngine is not a standalone TinyOS application, but a system component. Figure 7 represents the FSM generated by *FSMGen* for MultiHopEngine.

We can see that *FSMGen* is able to generate a compact as well as accurate FSM for MultiHopEngine. It is able to capture the behavior of the component when the `Send.send` command is called in state 4 by the application that uses this component. In state 0, the self-loops for the *Send.send* denote cases in which the message is not sent either because the packet size exceeds `TOSH_DATA_LENGTH`, or the node on which the application is running does not have a parent in the routing tree, or an attempt to send the packet on the radio fails. Only when the radio `SendMsg.send` succeeds does the component move into state 1 , where it waits for the `SendMsg.sendDone` event to occur.

MultiHopEngine provides a `Receive` interface for applications. An application programmer expects that an application running on the root node could use the `Receive` interface to receive packets sent up the tree. In fact, as we discovered by examining the FSM inferred by *FSMGen*, the MultiHopEngine implementation does not satisfy this expectation. We noticed that all edges for the `ReceiveMsg.receive` event in the FSM have a condition involving the calling of the `SendMsg.send` event, i.e. `SendMsg.send` is always called within the `ReceiveMsg.receive` event handler. This implies that for all packets received at any node, the packet is sent out on the network interface, never up to the application. Indeed, upon examining the application code, we found that at the root node, the packet is sent to the UART. A regular contributor to the TinyOS community expressed surprise at this finding, and we have verified that the TinyOS 2.x forwarding engine does not exhibit this behavior. We suspect that MultiHopEngine is always used with the root connected to a base station, and never with an application at the root node wired to MultiHopEngine.

FTSP To see how *FSMGen* performed on extremely complex programs, we ran it on the code for FTSP, a popular time synchronization protocol [23]. The code for FTSP contained around 46 branching conditions, of which 19 were part of the predicate abstraction. The FSM for FTSP before minimization had 255 states, and after using the unmodified Myhill-Nerode algorithm for more aggressive minimization, has 16 states (Figure 8).

We have verified that the FSM reflects the expected behavior but will refrain from discussing it since that requires a detailed description of the FTSP protocol. However, an interesting feature of the FSM is that all states have edges to state 6 on the event `ReceiveMsg.receive`, on the condition that although the node should be synchronized,

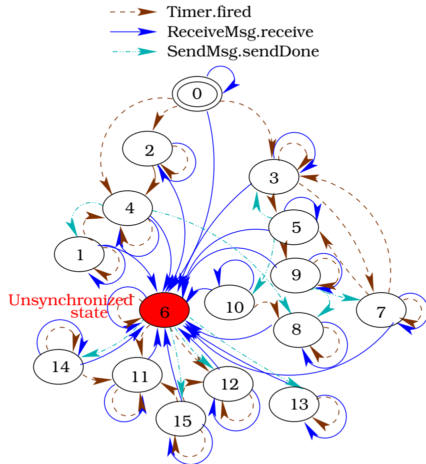

Figure 8. FSM for FTSP

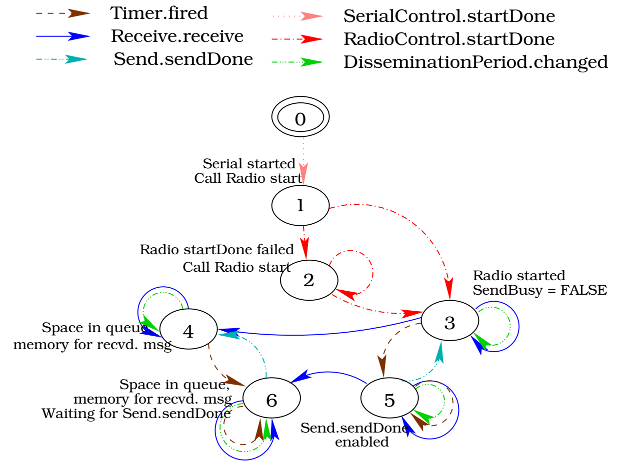

Figure 9. FSM for TestNetwork

the packet it receives has an error greater than the allowable error. This condition basically implies that the node has become unsynchronized and needs to clear its table of timestamp entries. Remarkably, *FSMGen* produces a single "error" state for this, even though this state is not evident when inspecting the code.

This FSM also illustrates a limitation of our symbolic execution framework, that we described in Section 4.2. In FTSP, when a node does not receive a beacon within a fixed number of timer events (five in the current implementation), the node sets itself as root and starts sending out beacons. Thus, the value of the predicate controlling this change in state, will only change after the `Timer.fired` event is fired for the fifth time. However, *FSMGen* simulates the `Timer.fired` event in this state only twice before stopping since it realizes that no new states are being introduced. Hence *FSMGen* is unable to capture this transition.

The FTSP example also illustrates another facet of *FSMGen*. It took nearly 24 hours for generating the FSM for FTSP. For us, this is not a cause for concern, for two reasons. First, *FSMGen* is not intended to be a frequently-used interactive tool. Rather, we expect programmers will use it occasionally when they make large-scale changes to system logic, or as part of a regression testing suite. Second, the bottleneck in FSM inference is symbolic execution, and well-known optimizations exist to scale this up to large programs [31]. We intend to implement these in a future version of the tool.

TestNetwork The TestNetwork application comes as part of the TinyOS-2.x distribution. It periodically sends packets up a collection tree rooted at the base station, using the Collection Tree Protocol (CTP). The sending rate is configurable via dissemination. We chose this application to demonstrate that *FSMGen* can easily be extended to TinyOS-2.x.

We generated an FSM for TestNetwork, which can be seen in Figure 9. While in TinyOS-1.x, the radio and other services are started using the simple `start()` function provided in their `StdControl` interface, in TinyOS-2.x, this `StdControl` interface has been replaced by the `SplitControl` interface. Thus, in TinyOS-2.x the start calls for Radio, Serial and other such devices form part of a split-phase operation. For example, calling `start` for the Radio component has to be followed by a `startDone` event whose argument is set to `SUCCESS`. If the argument is set to `FAIL`, that implies that the radio has not been started and that no packets can be received yet. By contrast, in TinyOS-1.x, after calling `start` we could safely assume that the radio was operational. Hence, we needed to add this domain-specific information to *FSMGen* in order to generate FSMs which captured this. We again used the aggressive version of our minimization algorithm to generate this FSM, since although the size of the TestNetwork code is reasonably small, we wanted to track the transitions due to *six* different events, the largest number of events in any of the programs we have used for this paper.

In Figure 9 we can see that the startup process for the application is captured quite nicely. Also, for transitions on `Timer.fired` we see that the program checks its state to see if the radio is busy, and then also checks to ensure that the `Send.send` function was called successfully. This is quite similar to the Surge application in TinyOS-1.x. The TestNetwork also receives packets over the radio, and if it has free space in its memory pool and the sending queue, pushes them in the queue to send over the serial port. If it has no space, it simply drops them.

We note here that for states 3 and 5, the `Receive.receive` event causes the program to

transition to other states if there is space in the message pool. However, for states 4 and 6, the program always remains in the same state upon a `Receive.receive` event. We verified by understanding the state machine and looking at the code, that there should be no difference in how `Receive.receive` is handled by any of the above states. This inconsistency is due to the fact that we disable our requirement on incoming edges during minimization and hence lose some information in the process. Ideally, for 4 and 6, if there was no space to store the packets, the program should have transitioned to a new state. However, using aggressive minimization was important here, as otherwise the number of states would have increased to 20. Thus there is a clear and obvious tradeoff between the size and readability of the state machines, and the overall functionality captured by *FSMGen*.

Summary. Thus, overall, *FSMGen* after being tested on a number of applications and other components, performed quite well, and was in a couple of cases, even able to capture inconsistencies in code written by others. It managed to generate a respectable state machine for a complex component like FTSP, and also worked well for TestNetwork, a TinyOS-2.x application with a large number of events.

6 Related Work

A number of works have supported and inspired our hypothesis that FSMs are good high-level abstractions of event-driven sensor network programs. Kasten et al. [17] present the design of OSM, a programming language that allows the programmer to directly implement sensor networks as finite state machines. Kim et al. [18] propose SenOS, a state-machine-based execution environment for sensor networks. We tackle a different problem than both these works, in that rather than trying to build a new programming architecture or operating system based on FSMs, we attempt to abstract programs written in the currently popular sensor network programming architectures into FSMs.

Other work requires FSM specifications to perform validation of sensor network programs. For example, Archer et al. [3] use FSMs to represent correct usage specifications of TinyOS interfaces, enforcing these specifications at run time. Lighthouse [28] uses FSM specifications in order to statically analyze dynamic memory usage in SOS [12] applications. Both of these systems require FSMs to be provided by the user. Our work could be used to infer FSMs as input to these and other kinds of static and dynamic analysis tools for sensor networks.

Tools for program analysis of sensor network programs have also been recently developed. cXprop [9] is an abstract interpreter built over CIL [24], designed for TinyOS. It allows users to define their own value-propagation analyses in the spirit of conditional constant propagation, using abstract value domains. cXprop contains a symbolic execution module, which manages the abstract values in the program state. cXprop uses a conservative concurrency model for nesC/TinyOS in order to track the state of shared variables. In contrast, we use an optimistic approximation of the concurrency and execution model of TinyOS, since our final goal is different from that of cXprop. Safe TinyOS [8] is a tool built using cXprop, which provides memory safety for TinyOS.

Within the programming languages community, there is a large body of related work on trying to derive FSMs from programs. Ammons et al. [2] profile multiple executions of an application and then employ machine learning on the resulting execution traces to infer an FSM. Static techniques are closer to our work. In particular, works by Alur et al. [1] and Henzinger et al. [13] employ forms of symbolic execution and predicate abstraction to infer FSMs. However, the goal of these works is to derive a *temporal specification* for a single component, which indicates the sequences of function calls that do not cause the component to crash (or throw an exception). This temporal specification can then be fed to automatic verification tools for checking clients of the component [6, 14]. Our differing goals lead to different design decisions. For example, they drive the construction of an FSM according to the ways in which the component can throw an exception, while we drive the construction of an FSM according to the application's control flow. Also, we perform minimization to make the resulting FSM user-readable, while this is not a concern for those works. Finally, we handle the event-driven and asynchronous constructs of TinyOS, while these works are implemented for mainstream languages (Java and C).

Symbolic execution is a very general technique and can be used for many different kinds of program reasoning. The paper by King et al. [19] is one of the earliest papers on the subject and describes the basic technique. Two recent systems employing symbolic execution similar to ours are ESP [10] and ARCHER [31]. ESP uses symbolic execution to check that code obeys a given temporal specification, and ARCHER uses symbolic execution to check for memory access errors in C programs. Similarly, a number of works [5, 4, 7] use predicate abstraction techniques in order to map an infinite (or very large) state space to a finite one. Typically this abstraction is used in order to perform a form of model checking on the resulting state space. None of these systems handles the novel features of TinyOS.

Finally, Jhala et al. [16] describe a formal algorithm and associated complexity results for inter-procedural dataflow analysis of programs with asynchronous calls/events. They also present some preliminary experience using the algo-

rithm to verify safety properties of programs. Their work could form the basis of a more formal analysis of our algorithm for symbolic execution of asynchronous programs, since symbolic execution can be viewed as a path-sensitive form of traditional dataflow analysis.

7. Conclusion

In this paper, we have tackled the problem of inferring compact, user-readable FSMs for applications and system components from TinyOS programs. Our *FSMGen* tool uses symbolic execution and predicate abstraction to statically analyze implementations in order to infer FSMs.

Our *FSMGen* tool uses a coarse approximation of the event-driven execution model of TinyOS, and hence the resulting FSM may not represent all possible execution paths. Through experiments, however, we have shown that this optimistic analysis provides FSMs that are both user-readable and detailed. We have tested *FSMGen* for a number of applications and system components and found that the inferred FSMs capture the functionality of the target applications quite well, and reveal interesting (potential) program errors. We suspect however, that this model may not be applicable to low-level interrupt driven code.

As future work, we plan to implement several optimizations to improve the efficiency of *FSMGen* and to apply the tool to larger TinyOS programs. We intend to investigate various widening techniques to improve how *FSMGen* deals with loops in TinyOS programs. We also plan to explore other uses of *FSMGen*. By improving our approximation of the TinyOS execution model, we might be able to detect scenarios, unforeseen by the programmer, which lead to race conditions in the execution of tasks and events, as described in Section 3.2. Also, *FSMGen*'s output could be used for automatic program verification; the Lighthouse memory checker, for instance, expects an FSM as input.

References

[1] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. *SIGPLAN Not.*, 40(1), 2005.

[2] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proc. of POPL*, 2002.

[3] W. Archer, P. Levis, and J. Regehr. Interface contracts for tinyos. In *Proc. of IPSN*, 2007.

[4] T. Ball, B. Cook, S. Das, and S. Rajamani. Refining approximations in software predicate abstraction. In *Proc. of TACAS*, 2004.

[5] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *Proc. of PLDI*, 2001.

[6] T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *Proc. of POPL*, 2002.

[7] M. Colon and T. E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Proc. of Computer Aided Verification*, 1998.

[8] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for tinyos. In *Proc. of SenSys*, 2007.

[9] N. Cooprider and J. Regehr. Pluggable abstract domains for analyzing embedded software. In *Proc. of LCTES*, 2006.

[10] M. Das, S. Lerner, and M. Seigle. Esp: path-sensitive program verification in polynomial time. In *Proc. of PLDI*, 2002.

[11] C. Frank and K. Römer. Algorithms for generic role assignment in wireless sensor networks. In *Proc. of SenSys*, 2005.

[12] C.-C. Han and et al. A dynamic operating system for sensor nodes. In *Proc. of MobiSys*, 2005.

[13] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *Proc. of ESEC/FSE*, 2005.

[14] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Proc. of SPIN*, 2003.

[15] J. Hill and et al. System architecture directions for networked sensors. *SIGOPS Oper. Syst. Rev.*, 2000.

[16] R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In *Proc. of POPL*, 2007.

[17] O. Kasten and K. Römer. Beyond event handlers: Programming wireless sensors with attributed state machines. In *Proc. of IPSN*, 2005.

[18] T. Kim and S. Hong. State machine based operating system architecture for wireless sensor networks. *Parallel and Distributed Computing: Applications and Technologies*, 2004.

[19] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7), 1976.

[20] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *Proc. of PLDI*, 2007.

[21] P. Levis and et al. The emergence of networking abstractions and techniques in tinyos. In *Proc. of NSDI*, 2004.

[22] P. Levis, D. Gay, and D. Culler. Bridging the gap: Programming sensor networks with application specific virtual machines. Technical Report UCB//CSD-04-1343, UC Berkeley, 2004.

[23] M. Maróti, B. Kusy, G. Simon, and Ákos Lédeczi. The flooding time synchronization protocol. In *Proc. of SenSys*, 2004.

[24] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. of CCC*, 2002.

[25] A. Nerode. Linear automaton transformations. *Proc. of the American Mathematical Society*, 9, August 1958.

[26] R. Newton, Arvind, and M. Welsh. Building up to macroprogramming: an intermediate language for sensor networks. In *Proc. of IPSN*, 2005.

[27] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *Proc. of SenSys*, 2005.

[28] R. 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, 2006.

[29] A. Stump, C. W. Barrett, and D. L. Dill. CVC: A oooperating validity checker. In *Proc. of CAV*, 2002.

[30] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *Proc. of EWSN*, 2005.

[31] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *Proc. of ESEC/FSE*, 2003.

[32] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. In *Proc. of SenSys*, 2007.