A Toolkit for Visualizing the Runtime Behavior of TinyOS Applications
Andrew R. Dalton, Jason O. Hallstrom
School of Computing, Clemson University
{adalton, jasonoh}@cs.clemson.edu

Abstract

TinyOS has proven to be an effective platform for developing reactive embedded network applications. However, the platform’s lean programming model and power-efficient operation come at a price: TinyOS applications are notoriously difficult to construct and debug. The development difficulties stem, in large part, from a programming model founded on events and deferred execution. In short, the model introduces non-determinism in the execution ordering of primitive actions (i.e., commands, events, and tasks). The resulting set of possible execution sequences is typically large, and can swamp developers’ unaided intellectual ability to reason about program behavior.

In this paper, we present a platform-neutral visualization toolkit for TinyOS 2.0 to aid in program comprehension. The goal is to assist developers in reasoning about the computation forest underlying a system under test, and the particular branches chosen during each run. The toolkit design includes (i) a full-featured static analysis and instrumentation library, (ii) a selection-based probe insertion system, (iii) a lightweight event recording service, (iv) a trace extraction and reconstruction tool, and (v) two visualization front-ends. We demonstrate the utility of the toolkit using standard system examples, and present an analysis of the toolkit’s resource usage and performance characteristics.

1 Introduction

Embedded network systems continue to receive tremendous attention across a wide range of application domains. The burgeoning community formed around this technology is a witness to its potential to transform the computing landscape. This new system class, with sensor networks serving as the typifying example, holds the potential to bring invisible computing from the pages of science fiction to the forefront of our daily lives. Indeed, the transformation has already begun: Sensor networks are changing the way we interact with our peers [9], observe our planet [20], and protect our communities [1] — and we are just getting started.

Our work is motivated by the observation that TinyOS [6,12] has served as an important catalyst for initiating this transformation, and by the expectation that it will continue to play a prominent role in the years ahead.

There are a number of factors that contribute to the continued — and arguably increasing — adoption of TinyOS.

Among the most significant is the programming model that the operating system provides to developers, and the benefits that this model imparts. In particular, the model aids developers in constructing applications that are both lightweight and power-efficient, making it well-suited to a host of hardware platforms and in-situ deployments1. The central guiding principle is a control flow architecture based on events and deferred execution. By contrast to the typical linear style of desktop- and server-class systems, TinyOS applications execute in a reactive style. Nodes spend much of their time sleeping, waking periodically to execute a short series of actions before falling back into an idle state. The short bursts of activity are triggered by hardware interrupts (e.g., timers, radio events) and the execution of previously deferred actions. Abandoning a thread-based model in favor of events limits memory and energy consumption — precious resources on platforms intended for in-situ deployment. But these benefits come at a significant price: TinyOS applications are notoriously difficult to construct and debug.

Problem Statement. The reactive model provided by TinyOS introduces non-determinism in the execution ordering of an application’s primitive actions. As a result, the number of control flow paths that must be considered to reason faithfully about the correctness and performance of a given system tends to be large. Equally important, this path set is usually difficult to identify through manual source inspection. This is in contrast to thread-based systems, in which the path set tends to grow slowly with program size, and tends to follow naturally from an application’s source structure. To illustrate this point, consider the three call graphs shown in Figure 1. The first, S1, corresponds to a standard thread-based system. The application includes a single entry point at X(). The path set for this system is captured by the regular expression, τS1, which generates 2 possible execution paths. Now consider S2, an event-based system without preemption, with an additional entry point

---

1While the contributions of this manuscript are platform-neutral, they are presented in the context of the Tmote Sky hardware platform [15]. Each matchbox-sized device includes an 8Mhz MSP430 microcontroller, 48K of instruction memory (ROM), 10K of data memory (RAM), 1Mb of off-chip EEPROM storage, and a 2.4GHz IEEE 802.15.4 (Zigbee) wireless transceiver. The platform is widely used in academic research, and is beginning to gain popularity in the commercial sector. Its hardware profile is representative of other standard platforms for embedded network construction. As we will see, accommodating the severe resource constraints of such platforms is a key component of the contributions.
at $Y()$. Assuming program actions execute atomically with respect to one another, the path set for this system is described by $\tau_{S2}$, which generates 3 possible execution paths. Finally, consider a TinyOS application, $S3$, in which event $Y$ may preempt event $X$. The corresponding path set, captured by $\tau_{S3}$, now includes 12 possible execution paths, and does not follow naturally from the system call graph. In real systems, the path sets are much larger; they can quickly swamp developers’ ability to reason about the behavior of their programs — especially in the “end cases”. The goal of our work is to provide tool support to assist developers in understanding the path sets underlying their systems, and the particular paths chosen during each run. We focus on a program visualization approach.

**Contributions.** We present a platform-neutral visualization toolkit for TinyOS 2.0 to aid in program comprehension. This is, to the best of our knowledge, the first toolkit of its kind for any sensor network platform. Two modes of operation are supported. The first is focused on static application structure, and consequently, on potential execution paths. The output in this mode consists of an annotated system call graph corresponding to a user’s source base. The second —and more interesting— mode of operation is focused on dynamic application behavior, and consequently, on actual execution paths. The output in this mode is an annotated UML sequence diagram corresponding to the behavior of a single application run. The toolkit architecture consists of five components; these components are the focus of this manuscript. First, we describe a full-featured static analysis and instrumentation library implemented in Java for TinyOS 2.0. Second, we describe a system constructed using this library to insert logging probes within a source base to capture the program actions of interest to a developer. Third, we describe a lightweight service for recording TinyOS execution events. Fourth, we describe a tool to extract logged execution events and to reconstruct the actual runtime trace. Finally, we describe two visual front-ends corresponding to the static and dynamic views introduced above. We demonstrate the utility of the toolkit using standard system examples included as part of the TinyOS 2.0 distribution. We also present a detailed analysis of the toolkit’s resource usage and performance characteristics.

**Paper Organization.** The remainder of the manuscript is organized as follows: Section 2 surveys some of the most important elements of related work and highlights the novelty of our contributions. Section 3 presents a brief overview of the TinyOS operating system and the nesC programming language. Section 4 details the design and implementation of the visualization toolkit. Section 5 presents use-case scenarios that demonstrate the toolkit’s utility in the context of standard TinyOS programs. Section 6 presents an analysis of the toolkit’s performance and associated runtime overhead. Finally, Section 7 concludes with a summary of contributions.

2 Related Work

That sensor network systems are difficult to construct and debug is hardly a new observation. The exploration of techniques and tools for reducing this difficulty continues to be a major research thrust within the community. Here we survey some of the most significant efforts reported in the literature. We additionally survey key results in program visualization. While this topic has received relatively little attention in the domain of sensor networks, it has a long and rich history in the domain of desktop systems.

**Fault Localization.** Several authors have described techniques for detecting and localizing faults in sensor networks. Ramanathan et al. [16] present an approach based on comparisons between actual and expected network traffic patterns. While helpful in identifying node-level fault candidates, the approach does not aid in identifying source-level problems. In contrast, Krunic et al. [8] focus on providing source-level assistance. The authors describe a diagnostic system designed to trap program faults before they can disable the hosting device. The system includes a network interface for collecting context information related to a fault, including runtime trace information. Our approach to encoding trace information using numeric tokens is similar to the approach discussed in [8].

Improving runtime visibility has been another important focus in the literature. Tolle and Culler [19] describe a network management system that enables developers to expose attributes as part of a program implementation. Attributes are encoded manually, and can be read and written across a network at runtime from a basestation. The system additionally provides the ability to log events at manual instrumentation points, and to exfiltrate log data for later analysis. Again, the event encoding approach is similar to ours. Whitehouse et al. [21] extend this work; they describe a system for accessing program state without the need for manual

---

2This technique appears to represent a new software design pattern; it bears some similarity to the Flyweight pattern originally codified in [4].
attribute encoding. The toolkit additionally provides remote procedure call capabilities from a basestation to a network node. Dalton and Hallstrom [2, 3] describe a sensor network testbed that leverages the work of Tolle and Culler to provide state visibility across a test network. The system also provides automated exfiltration and interpretation of network message streams. While these systems have proven useful in improving runtime visibility, they provide little insight into the path sets underlying a system, or the particular paths chosen during a run — the focus of our work.

Program Visualization. A myriad of program visualization tools for desktop systems have been discussed in the literature. Here we consider three representative efforts.

Jones et al. [7] describe an approach to visualizing program execution data collected from deployed software. Their focus is on developing visual abstractions that scale to large datasets, and that can be tailored for use across different types of execution data. The approach includes three visual abstractions used to represent statement-, file-, and system-level views of a collected dataset, respectively. The authors also describe approaches to filtering and summarizing runtime data across multiple runs. While the utility of the authors’ work has been vetted in a number of contexts, none of the three visual abstractions are well-suited to reasoning about program control flow. Our work, on the other hand, is specialized for this purpose.

Closer to our work is that of Lange and Nakamura [10]. The authors describe a program visualization toolkit for C++ applications that combines static structural information and dynamic trace information to generate object-centric views of program behavior. The toolkit components are similar to ours; they include an instrumentation system, an execution trace recoder, and a program database from which static program information can be retrieved. The generated visual representations include object creation and lifetime graphs, as well as object-centric call graphs. A simple selection interface enables filtering on classes and methods. Malloy and Power [13] describe a similar, but more advanced visualization system for C++ applications. The visualized components include class and method call graphs, UML communication diagrams, and UML sequence diagrams. The system additionally includes static and dynamic event filtering, and supports dynamic visualization during the execution of a program under test. In contrast to the work of these authors, our work targets TinyOS applications. The programming model (and associated language) is fundamentally different than that provided by C++. Moreover, our approach operates in an asynchronous execution environment under tight resource constraints.

Finally, it is worth noting that a number of groups have recently released development environments for TinyOS [14, 17, 18]. The environments include a subset of the features found in standard development environments, including syntax highlighting, automated code completion, compilation support, etc. The environments additionally provide support for visualizing the static structure of TinyOS applications. The representations range from simple hierarchical component views to more detailed representations of component bindings and call graph structure. The tools do not, however, support visualization of dynamic program behavior, which is our main focus.

3 Background: TinyOS and nesC

TinyOS applications are implemented in nesC [5, 11], a component-based dialect of the C programming language. Applications are constructed by composing application-specific components with general-purpose components provided by the operating system (e.g., timers, device drivers, network protocols). Here we summarize the basic language constructs and execution model. We omit some advanced language features (e.g., generic programming, parameterized interfaces, single function wirings), but note that these features are supported by the visualization toolkit.

In nesC, the basic programming abstraction is the component\(^7\), which bears some similarity to a class. Components encapsulate state and behavior, and expose this behavior through interfaces. An interface consists of command and event declarations. A command is analogous to a class method; it is a function that can be invoked on a providing component. In contrast, an event is a function invoked by a providing component. An event declaration specifies the signature of the handler that will be invoked when the event is signaled. For example, Timer\_fired() specifies the signature of the handler invoked when an underlying timer event is signaled. Notice that commands impose an implementation responsibility on providing components, whereas events impose an obligation on using components.

In addition to exposing public behavior, components may implement private behavior. Private behavior is realized by functions accessible only from within the declaring component. A component may include standard C style functions, as well as one or more tasks. A task is a special type of function posted for later execution.

One of the most interesting characteristics of nesC is that it enforces loose component coupling. The language does not allow a component to explicitly name the target of a command or event invocation. Instead, all invocations must flow through interfaces, without naming the components that provide those interfaces. Hence, components not only specify the interfaces they provide, but the interfaces they use. Interface dependencies are resolved by configurations that wire interface users to interface providers.

Control flow originates from one of two sources. The first source is the TinyOS task scheduler. Call chains orig-
5.1 nesC Analysis and Inst. Library

The visualization toolkit is implemented using the nesC Analysis and Instrumentation Library, a general-purpose Java API for parsing, analyzing, instrumenting, and generating nesC source code. Although the visualization toolkit is among the first applications to use the library, the API is designed to enable a broad class of program analysis and instrumentation tools. We believe that the library provides a foundation for expanding software engineering and programming languages research in the context of TinyOS.

The most fundamental feature provided by the API is the ability to parse a nesC source base. While it is possible to process individual files, many applications benefit from a configuration-based parse: The API provides a method to initiate a parse from a specified configuration. The method accepts the configuration path as argument, and a list of search paths used to locate dependent components identified during the parse. The result is a set of abstract syntax trees (ASTs) corresponding to the translation units processed during the parse. These trees can be processed manually using standard accessor methods. Alternatively, the API provides a set of convenience methods to simplify the most common tasks. While a full treatment of the library is beyond the scope of this paper, a small sample of the methods provided by the library are included in Table 1.

The table is divided into three sections corresponding to the three basic types of methods provided by the API. The first section lists two example traversal methods. getNodesOfType() returns a collection of nodes that satisfy a given

---

**Figure 2. Visualization Process Overview**

In the more interesting case, the metadata is used as input to generate an annotated call graph for the system under test. In the simple case, this metadata can be used to locate dependent components identified during the parse. The result is a set of abstract syntax trees (ASTs) corresponding to the translation units processed during the parse. These trees can be processed manually using standard accessor methods. Alternatively, the API provides a set of convenience methods to simplify the most common tasks. While a full treatment of the library is beyond the scope of this paper, a small sample of the methods provided by the library are included in Table 1.

The table is divided into three sections corresponding to the three basic types of methods provided by the API. The first section lists two example traversal methods. getNodesOfType() returns a collection of nodes that satisfy a given

---

4. Visualization Toolkit

We now turn our attention to the design and implementation of the visualization toolkit. The steps involved in applying the toolkit are summarized in Figure 2. The first step is to analyze and transform the source base of the system under test. The output of this step is an instantiated source base free of generic types, and a collection of metadata detailing program symbols and function calling relationships. In the simple case, this metadata can be used to generate an annotated call graph for the system under test. In the more interesting case, the metadata is used as input to a second step, in which a developer selects a set of functions to be traced. The selected functions are used to guide the insertion of probes that record enter and exit events on the functions of interest. The output of this step is an instrumented source base and a symbol map on the functions of interest. The output of this step is an entry that enables the second source of control flow. Call chains originating from an interrupt are said to execute in an asynchronous context. An asynchronous call chain may preempt a synchronous call chain, or even preempt an active asynchronous chain. Commands and events executed along an asynchronous chain typically require special logic to prevent race conditions, whereas synchronous call chains typically do not.

5 This rule is not strictly enforced; violations are treated as warnings. Violations are, however, considered to be dangerous, and rarely occur in deployed systems. We ignore this possibility in the design of our toolkit.

6 The full nesC grammar is supported, as implemented by version 1.2.7a of the nesC compiler.
<table>
<thead>
<tr>
<th>Type</th>
<th>Method Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traversal</td>
<td>getNodesOfType(), getParent(), ...</td>
</tr>
<tr>
<td>Generation</td>
<td>generateExprStmt(), generateEnum(),</td>
</tr>
<tr>
<td></td>
<td>instantiateGenerics(), ...</td>
</tr>
<tr>
<td>Modification</td>
<td>addComponentToConfig(),</td>
</tr>
<tr>
<td></td>
<td>instantiateGenerics(), ...</td>
</tr>
</tbody>
</table>

Table 1. A&I Library Methods (partial)

Type specification based on a full traversal of the AST rooted at the invocation target. This method is useful, for example, in identifying the modules referenced by a configuration. The `getParent()` method has the obvious meaning, and is useful in discovering the containing context associated with nodes returned by `getNodesOfType()`.

The second section lists two example code generation methods. `generateExprStmt()` accepts an arbitrary nesC expression as argument (in string form), and returns the corresponding AST. This method is useful, for example, in generating the probes inserted to capture runtime trace data. `generateEnum()` is similar; the method is used to generate enum declarations. The method is used by the toolkit to introduce unique identifiers for instrumented modules.

The final section lists two example code modification methods. `addComponentToConfig()` is used to introduce a new component reference into an existing configuration. This is useful, for example, in injecting the recording service required to capture trace information. The second method, `instantiateGenerics()`, is among the most complex operations provided by the API. The method transforms a set of ASTs to eliminate generic components and generic configurations. The approach is to duplicate the generic types by substituting actual arguments for formal generic parameters. The instantiated components are automatically renamed to include a unique integer tag to prevent collisions. All component references are updated to reflect the new names. This method has proven extremely valuable in developing the visualization toolkit, where component instances must be differentiated and instrumented separately.

### 4.2 Probe Selector

From a user’s perspective, the Probe Selector—or simply Selector—is the first application in the visualization tool chain. The Selector expects the top-level configuration of the target application to be passed as argument. On startup, the A&I Library is used to parse the input system and to generate a listing of the components it defines, as well as the associated commands, events, tasks, and functions. This list is presented to the user, who then selects the program actions to be included in the visualization.

When the actions of interest have been selected, the A&I Library is used to apply two system transformations. First, the source base is instantiated to eliminate generic components and generic configurations. This step is required to enable users to differentiate control flow across instances of the same generic component. If, for example, a system includes two instances of TimerMilliC, the instantiation process will create two new components, each suitably renamed, by instantiating TimerMilliC. Metadata describing the program symbols and function calling relationships is stored for later use in generating an annotated call graph.

The second transformation involves injecting logging probes at the entry and exit points of the selected actions. The basic instrumentation procedure is illustrated by the code fragments shown in Listing 1. The first listing shows a simple nesC command prior to instrumentation; the second shows equivalent instrumented code. The body of each instrumented action is wrapped within an anonymous block, and the `enter` event is recorded before the block. This allows probes to be injected before variable declarations (which might involve function calls). Similarly, each exit point is wrapped within a statement-expression to capture `exit` events before the action terminates. In general, multiple exit points may need to be instrumented.

Note that each probe records the `instanceId` used to identify the containing component. This identifier is introduced as a module-level enumeration constant during the instrumentation process. Also note that a second constant is used to identify the containing action. The generated constants are mapped to the corresponding signatures and exported as a symbol map for later use in reconstructing a recorded trace. It is worth noting that the underlying storage structure used to log program events is generated dynamically. This is done to minimize the number of bits required to uniquely identify the selected modules and actions.

### 4.3 Event Recording Service

The Event Recording Service is implemented as a single component, TraceRecorderC, which serves as a thin wrapper over the standard log storage component provided by TinyOS (LogStorageC). The component provides a simple interface, Trace, which defines commands to log `entry` and `exit` events. (The component is initialized at system startup.) Internally, the component implements a dual

Listing 1. Instrumentation Procedure

```c
// PRE-TRANSFORMATION:
command message_t MessageQueue.dequeue() {
uint8_t size = call Queue.size(); ...
return Queue.dequeue(); }

// POST-TRANSFORMATION:
command message_t MessageQueue.dequeue() {
call Trace.enter(instanceId, 1); {
uint8_t size = call Queue.size(); ...
return ((message_t *nesctk_return;
nesctk_return = Queue.dequeue();
call Trace.exit(instanceId, 1);
nesctk_return; }); } }
```
buffering strategy. A current buffer is used to cache program events logged to the trace. When the current buffer becomes full, the second buffer is swapped into its place while the first buffer is flushed⁶. This cyclic process repeats to prevent missed events during log storage updates. We note that in our testing of TraceRecorderC, 70 entries of 3 bytes each is close to the maximum buffer size that can be consistently written to log storage without bit errors.

4.4 Trace Extractor / Collector

After a run, the trace data must be extracted and reconstructed for use in generating the corresponding sequence diagram. This is achieved by installing a new application image on the device. The Trace Extractor retrieves the captured data from log storage and transmits the data to an attached basestation. The Trace Collector is used at the basestation to receive the transmitted data and to reconstruct the trace. Reconstruction involves mapping the module and action identifiers stored within the trace back to the corresponding module names and action signatures. This is done using the symbol map generated by the Selector. The reconstructed trace is then saved for later use.

4.5 Call Graph Generator

The Call Graph Generator is used to visualize an application’s static call graph. The interface is similar to that of the Selector. At startup, the A&I Library is used to perform a full parse of the target application based on the top-level configuration passed as argument. The user is then prompted to select the program actions of interest. This selection step is designed to focus the generated graph on a manageable subset of program actions. (We introduced this step after our initial experiences visualizing call graphs with thousands of actions.) The generated graph includes the selected actions, actions that invoke the selected actions, and actions invoked by the selected actions. As we will see, this selection mechanism provides an effective means of limiting the scope of the graph while retaining suitable detail to reason about the actions of interest.

4.6 Sequence Diagram Generator

The Sequence Diagram Generator is used to transform a reconstructed trace into a corresponding sequence diagram. The generated diagram follows standard UML conventions, with minor adaptations to suit the semantics of nesC. The object rectangles that traditionally appear at the top of the diagram are used to represent component instances. The activation rectangles running vertically have the usual meaning; they represent activations of the corresponding component. Solid arrows between activations represent invocations, and dashed arrows represent returns. It is important to note that edges capture transitive invocation relationships. A call chain from action A() to B() to C() will be represented by an edge from A() to C() if B() is outside the instrumentation set of the system under test.

To distinguish between commands, events, tasks, and module functions, we use the following coloring scheme: Orange, purple, blue, and green activation rectangles are associated with commands, events, tasks, and module functions, respectively. Similarly, to enable users to distinguish between synchronous and asynchronous actions, we use an edge coloring scheme. Black and red edges are used to represent calls to synchronous and asynchronous actions, respectively. Blue edges are used to represent calls to local functions, which can be invoked from either a synchronous or asynchronous context. (We note that edge color is based on static signature data; determining the runtime context of a call will typically require probes outside the system’s instrumentation set.) We will see that this coloring scheme improves the utility of the sequence diagram notation in supporting developers’ understanding of system behavior.

5 Use-Case Scenarios

At this point it is useful to consider two scenarios that illustrate the use of the toolkit and the benefits that it provides. The scenarios involve standard application examples included as part of the TinyOS 2.0 distribution: Blink and RadioCountToLeds. Despite their lack of surface complexity, these applications are rich with interesting behavior.

5.1 Scenario 1: Blink

The scenario begins with a developer interested in investigating the timing behavior of Blink. As a starting point, she may choose to view the system call graph by invoking the Call Graph Generator, passing the top-level configuration file, BlinkAppC.nc, as argument. After selecting VirtualizeTimerC.fireTimers() as the focal point of the visualization, the view shown in Figure 3 is displayed. At a
glance, the developer might take an interest in the fact that all system timers are being dispatched from `fireTimers()`.

To investigate this behavior further, she may choose to visualize the execution of Blink. She first selects the program actions to be traced using the **Probe Selector** shown in Figure 4, again passing `BlinkAppC.nc` as argument. In the figure, she has already selected several actions. When the selection process is complete, the instrumented source base is generated, and the corresponding symbol map is exported for later use. The system is then compiled and installed.

To collect the resulting trace data, the **Trace Extractor** is installed on the target device, and the **Trace Collector** is executed to reconstruct the runtime trace. The symbol map is passed as argument at startup. Finally, the reconstructed trace is passed to the **Sequence Diagram Generator** to produce the diagram shown in Figure 5.

The diagram captures a canonical example of device virtualization in TinyOS. The first execution sequence begins with an asynchronous event, `Alarm.fired()`, signaled on `AlarmToTimerC`. The event originates from an actual clock source. (The event source is designated as System since the actual signaling action is outside the instrumentation set of Blink.) This event posts a task, `fired()`, which is the next action executed in the sequence. The task signals `Timer-From.fired()` on `VirtualizeTimerC`, which in turn invokes a local dispatching function, `fireTimers()`. The dispatching function signals `fired()` on all pending virtual timer instances. Finally, these events trigger the main BlinkC module to invoke LED toggle functions on `LedsP`. In the first series of program actions, all the virtual timers used by Blink are pending, so each `fired()` event is signaled. In the second series, only two of the virtual timers are pending (i.e., `Timer1` and `Timer0`); hence there are only two `fired()` events reflected in the diagram. We note that it can be difficult to reason about the behavior of a virtualized device based on manual inspection of the program source code. By contrast, the behavior is clear from the sequence diagram.

The third execution sequence is also interesting, and reveals a minor modification of Blink introduced for testing purposes. Specifically, Blink was modified to include an asynchronous event triggered when the user button is clicked on the hosting device. As shown in the diagram, this event interrupted the (synchronous) `Timer3.fired()` event. While the example is obviously contrived, it is representative of a larger class of behaviors — behaviors that can be difficult to understand without a visualization tool.

5.2 Scenario 2: RadioCountToLeds

The second scenario is focused on a developer interested in the runtime behavior of RadioCountToLeds. We omit the individual steps in the visualization process, and skip to the generated sequence diagram shown in Figure 6.

---

7The figures in this section were generated using the Tmote Sky platform from Moteiv. All figures have been trimmed and condensed for the sake of presentation. Solid horizontal breaks denote event omissions.
Once again the generated diagram captures interesting behavior that could otherwise be difficult to understand. In particular, the diagram illustrates active message filtering and dispatch in the CC2420 radio stack. Three call chains are captured, each triggered by the receipt of a message. Each chain begins with the execution of receiveDone_task(), a task implemented by CC2420ReceiveP. This task is responsible for populating elements of the message header before signaling the SubReceive.receive() event on UniqueReceiveP. Within UniqueReceiveP, hasSeen() is invoked to determine if the message is a duplicate of a previous message. The behavior illustrated in the figure indicates that the received messages were not duplicates; SubReceive.receive() is consequently signaled on CC2420ActiveMessageP in each chain. This event is responsible for filtering messages based on the destination address, and performing dispatch based on the active message identifier within the message. AMPacket.isForMe() is invoked to determine whether the message is addressed to the hosting node (or intended for all nodes). In the first call chain, the message is intended for the hosting device, and Receive.receive() is signaled on RadioCountToLedsC, which updates the state of the host’s LEDs based on the content of the message. In the second call chain, AMPacket.isForMe() returns FALSE, and the message is silently discarded. In the final call chain, the message is intended for the hosting device, but the specified active message identifier does not have an associated handler within the application. In this case, the default handler defined in CC2420ActiveMessageP is executed. We note that this diagram helped us to resolve errors in our own understanding of the radio stack’s behavior.

### 6 Experimental Analysis

We now examine the performance characteristics of the visualization toolkit, and the resource requirements it imposes on target systems. The focus is on evaluating the rate at which program events can be captured, and the overhead introduced by the instrumentation process.

#### 6.1 Capture Rate

The capture rate of the event recording service plays an important role in assessing the toolkit’s utility. If the maximum capture rate is low, the toolkit will not be suitable for fine-grained visualization, or for systems in which instrumented actions will be executed with high frequency. To evaluate the capture rate, we developed a test application to measure the recording time over a continuous stream of events. The application repeatedly logs a full event buffer (i.e., 70 events), and records the duration of each call. The process is repeated for three different event record types, ranging in size from 1 byte to 3 bytes. This is to account for the fact that event records are dynamically sized based on the instrumentation set to minimize storage requirements. The tests were performed using the Tmote Sky platform.

A summary of the experimental results is shown in Table 2. Each row corresponds to a single run of the test application using event records of the specified size. Buffer Count indicates the number of buffers written to log storage before the target volume reached capacity. The remaining columns have the obvious meanings. Despite the large variation in recording time witnessed during each run, the results are favorable. In the worst case, it took 1.058 seconds to log 70 records of 3 bytes each. Hence, given the dual-buffer implementation, the recording service can handle...
approximately 70 events per second, independent of the inter-arrival rate. (We note that the maximum capture rate can be tuned by increasing the number of event buffers, but at the expense of additional overhead.) This makes the toolkit especially well-suited to the visualization of bursty execution patterns, in which a node periodically wakes to perform a dense series of actions, and then resumes its idle state. Fortunately this pattern is representative of most sensor network applications.

### 6.2 Resource Overhead

The resource overhead introduced during the instrumentation process is another important evaluation metric. If the overhead is high, the achievable instrumentation density will be low, resulting in low-fidelity visualizations. This would also limit the potential integration outlets for the toolkit (e.g., existing sensor network diagnostic systems). To evaluate the impact on resource usage, we applied the toolkit to each of the sample applications included as part of the TinyOS 2.0 distribution. The instrumented systems were used to evaluate the base and incremental impact of probe insertion on application memory usage.

To distinguish between the overhead introduced by LogStorageC and the overhead introduced by the full recording service (of which LogStorageC is a part), each application was compiled under three configurations. In the first configuration, no source modifications were performed. In the second, LogStorageC was included in the application image, as were calls to erase the log and record a single event. (The calls were introduced to prevent the compiler from removing LogStorageC as part of its dead code elimination phase.) Finally, in the third configuration, the application was modified to include the full recording service, including the necessary calls to prevent elimination.

The resource requirements under each configuration are shown in Table 3. Application names have been abbreviated for the sake of presentation. The Baseline columns correspond to the first configuration; the EEPROM Logging columns correspond to the second configuration; and the Full Inst. Logging columns correspond to the third configuration. The RAM results are summarized in Figure 7, and the ROM results are summarized in Figure 8. The results are consistently favorable. On average, the instrumentation toolkit introduces a base cost of approximately 256 bytes of RAM and 6239 bytes of ROM. (Again, the size of the logging buffers can be tuned to reduce RAM overhead, but at the expense of reducing the maximum achievable capture rate.) Factoring out the overhead introduced by LogStorageC yields the additional cost of the recording service in applications that already include LogStorageC: 147 bytes of RAM and 802 bytes of ROM, on average. We view this overhead as acceptable for most applications.

It is also important to consider the incremental cost of each probe. In general, the instrumentation cost of a given action varies based on the number of exit paths it contains, as well as the compiler optimization context in which the probes appear. To give a sense of the typical cost associated with a single probe, we compiled several versions of the Blink application, increasing the number of probes from one version to the next. The first version contained a single entry probe; the second contained an entry probe followed by an exit probe; the third added a second entry probe; etc. The underlying records used to store events were 1 byte each. The resource requirements are summarized in Figure 9. Again, the results are favorable. While there is some variation in incremental cost, a new probe requires approximately 12 additional bytes of ROM on average. There is no incremental RAM expense, assuming that an additional probe does not increase the minimum required size of the underlying event records. Again, the resource requirements appear to be acceptable for most applications.

### 7 Conclusion

Our work began with an interesting observation. The efficiency characteristics that drive the continued adoption of TinyOS [6, 12] stem from the same source responsible for the platform’s disrepute among developers — a programming model based on events and deferred execution. While well-suited to the construction of lightweight, power-

<table>
<thead>
<tr>
<th>Application</th>
<th>Baseline</th>
<th>EEPROM</th>
<th>Full Inst.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RAM</td>
<td>ROM</td>
<td>RAM</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Logging</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RAM</td>
</tr>
<tr>
<td></td>
<td>1610</td>
<td>15160</td>
<td>1766</td>
</tr>
<tr>
<td>Blink</td>
<td>55</td>
<td>2654</td>
<td>216</td>
</tr>
<tr>
<td>MVis</td>
<td>1722</td>
<td>33640</td>
<td>1854</td>
</tr>
<tr>
<td>MHOsc</td>
<td>3368</td>
<td>28624</td>
<td>3498</td>
</tr>
<tr>
<td>MHOscLqi</td>
<td>2451</td>
<td>24438</td>
<td>2589</td>
</tr>
<tr>
<td>Null</td>
<td>97</td>
<td>1418</td>
<td>184</td>
</tr>
<tr>
<td>Oscope</td>
<td>372</td>
<td>15320</td>
<td>504</td>
</tr>
<tr>
<td>Powerup</td>
<td>4</td>
<td>1472</td>
<td>186</td>
</tr>
<tr>
<td>RadCnt2Lds</td>
<td>286</td>
<td>10794</td>
<td>420</td>
</tr>
<tr>
<td>RadSst2Lds</td>
<td>344</td>
<td>15158</td>
<td>474</td>
</tr>
<tr>
<td>Sense</td>
<td>95</td>
<td>7184</td>
<td>256</td>
</tr>
</tbody>
</table>

**Table 3. Base Overhead (RAM/ROM)**

**Figure 9. Incremental Overhead (ROM)**

*Full Instrumentation* columns correspond to the third configuration. The RAM results are summarized in Figure 7, and the ROM results are summarized in Figure 8. The results are consistently favorable. On average, the instrumentation toolkit introduces a base cost of approximately 256 bytes of RAM and 6239 bytes of ROM. (Again, the size of the logging buffers can be tuned to reduce RAM overhead, but at the expense of reducing the maximum achievable capture rate.) Factoring out the overhead introduced by LogStorageC yields the additional cost of the recording service in applications that already include LogStorageC: 147 bytes of RAM and 802 bytes of ROM, on average. We view this overhead as acceptable for most applications.
efficient systems, the model introduces non-determinism in the execution order of program actions. The result is an explosive increase in the number of execution paths that must be considered to reason faithfully about an application’s correctness and performance. In addition to a burgeoning path set, developers must contend with hidden paths that do not follow naturally from manual source inspection. The complexity can quickly swamp developers’ ability to understand the path sets underlying their systems, and the particular paths chosen during each run.

To assist developers in performing these reasoning tasks, we described a platform-neutral program visualization toolkit for TinyOS 2.0. The toolkit is used to generate the annotated call graph corresponding to an input system, and UML sequence diagrams corresponding to particular runs of the system. The toolkit architecture consists of five components. First, we described a Java-based static analysis and instrumentation library for TinyOS 2.0. Second, we described a system to automate the insertion of source-level logging probes to capture a desired set of program actions. Third, we described a simple, lightweight logging service for capturing runtime events. Fourth, we described a tool for extracting logged events and reconstructing the associated runtime trace. Finally, we described two visualization front-ends corresponding to the aforementioned views. The application of the toolkit was demonstrated using system examples included as part of the standard TinyOS 2.0 distribution. A detailed analysis of the toolkit’s resource usage and performance characteristics was also presented.

Acknowledgments. This work was funded in part by grants from the National Science Foundation (CNS-0520222, DUE-0633506). The authors gratefully acknowledge the NSF for its support.

References