Visualizing the Runtime Behavior of Embedded Network Systems

Andrew R. Dalton\textsuperscript{a}, Sally K. Wahba\textsuperscript{b}, Sravanthi Dandamudi\textsuperscript{b}, Jason O. Hallstrom\textsuperscript{b,}\textsuperscript{*}

\textsuperscript{a}Department of Mathematics and Computer Science, Western Carolina University, Cullowhee, NC 28723, USA
\textsuperscript{b}School of Computing, Clemson University, Clemson, SC 29634, USA

Abstract

TinyOS has proven to be an effective platform for developing lightweight embedded network applications. But the platform’s lean programming model and power-efficient operation come at a price: TinyOS applications are notoriously difficult to construct, debug, and maintain. The development difficulties stem largely from a programming model founded on events and deferred execution. In short, the model introduces non-determinism in the execution ordering of primitive actions — an issue exacerbated by the fact that embedded network systems are inherently parallel, distributed, and reactive. The resulting set of possible execution sequences for a given system 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 supports comprehension activities involving both local and distributed runtime behavior. The constituent components include (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) three visualization front-ends. We demonstrate the utility of the toolkit using both standard and custom system examples and present an analysis of the toolkit’s resource usage and performance characteristics.

Key words: Embedded networks, sensor networks, TinyOS, program visualization, distributed behavior, dynamic analysis, static analysis, program comprehension

* Corresponding author.

Email addresses: adalton@wcu.edu (Andrew R. Dalton), sallyw@cs.clemson.edu (Sally K. Wahba), sdandam@cs.clemson.edu (Sravanthi Dandamudi), jasonoh@cs.clemson.edu (Jason O. Hallstrom).

1 Introduction

Embedded network systems continue to attract tremendous interest across a diverse 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 [2,16], observe our planet [14,31,42], and protect our communities [1,37,40] — and we are just getting started. Our work is motivated by the observation that TinyOS [13,20] 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. Specifically, 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 deployments. The central guiding principle is a control flow model 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 —typically involving coordination with peers— before falling back into an idle state. The short bursts of activity are triggered by internal interrupts (e.g., clock sources, ADC completion signals), external interrupts (e.g., message reception events), and the execution of previously deferred actions (e.g., long-running operations best executed outside interrupt context). 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, maintain, and debug.

Problem Statement. The reactive model provided by TinyOS introduces non-determinism in the execution ordering of an application’s primitive actions. This non-determinism is further increased by the application contexts in which TinyOS is applied: Embedded network applications are inherently parallel and distributed. As a result of the programming model and (typical) programming context, the number of control flow paths that must be considered to reason faithfully about the correctness and performance of a given network system tends to be large. Further, this path set is usually difficult to identify through manual source inspection, especially in the presence of
Fig. 1. Example Execution Patterns

distributed interactions and environment-triggered behaviors.

While the non-determinism associated with parallel and distributed systems is well known, the non-determinism inherent to the TinyOS programming model requires some motivation. In contrast to thread-based systems, in which path sets tend to grow slowly with program size and follow naturally from application structure, path sets grow exponentially in TinyOS systems and tend not to mirror source structure. To illustrate this point, consider the three call graphs shown in Figure 1. The first, $S_1$, 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, $\tau_{S_1}$, which generates 2 possible execution paths. Now consider $S_2$, an event-based system without preemption, with an additional entry point at $Y()$. Assuming program actions execute atomically with respect to one another, the path set for this system is described by $\tau_{S_2}$, which generates 3 possible execution paths. Finally, consider a TinyOS application, $S_3$, in which event $Y$ may preempt event $X$. The corresponding path set, captured by $\tau_{S_3}$, now includes at least 12 possible execution paths, depending on the preemption policy (i.e., allowable depth, preemption priorities, and reentrancy constraints). Further, the execution paths do not follow naturally from the system call graph. In real systems involving asynchronous message traffic and environment-triggered events, 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 that supports comprehension activities involving both local and distributed runtime behaviors.
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. Three 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 latter—and more interesting—modes of operation focus on dynamic application behavior, and consequently, on actual execution paths. The output in each case is an annotated UML sequence diagram corresponding to the behavior of a single application run. In the first of these dynamic modes, the generated diagrams record local interactions among participating components. In the second, the diagrams capture distributed interactions in a manner that highlights causally-connected events within a network and demarcates regions of (logically) concurrent activity.

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\(^1\). 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 recorded events across participating nodes and to reconstruct the associated runtime trace(s). Finally, we describe three visualization front-ends corresponding to the static and dynamic views introduced above. We demonstrate the utility of the toolkit using standard network system examples included as part of the TinyOS 2.0 distribution, as well as a custom example involving distributed coordination. We also present a detailed analysis of the toolkit’s resource usage and performance characteristics.

Before proceeding, we note that while the contributions of this manuscript are platform-neutral, they are presented in the context of the Tmote Sky hardware platform [26]. Each matchbox-sized device consists of 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.

\(^1\) We limit our presentation to a survey of the library’s basic functions, referring the reader to [8] for a complete treatment. This technical report is new, authored after the publication of the original ICPC manuscript.
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 and pointers to future work.

2 Related Work

That sensor network systems are difficult to construct and debug is hardly a new observation. And while reducing this difficulty continues to be a significant research driver within the community, our work appears to be the first to consider a program visualization approach. Program visualization does, however, have a rich history in the domain of desktop systems. Here we survey some of the most significant efforts reported in the literature. We conclude with an overview of program comprehension projects for sensor network systems (which do not involve a visualization component).

2.1 Visualizing Local Runtime Behavior

Myriad approaches to visualizing local runtime behavior have been proposed. A key focus has been the development of visual abstractions that support program comprehension activities involving large trace datasets. Hamou-Lhadj et al. [12], for example, describe an Eclipse plugin that supports interactive exploration of a recorded trace sequence. Each sequence is represented using a standard “tree-view”. The main challenge is to ensure intellectual scalability of the view across large datasets. To this end, the authors describe filtering mechanisms that collapse subtrees based on recurring subtree patterns, depth culling, relaxed call ordering relations, and other tree transformations. A selection-based filtering mechanism is also discussed.

Cornelissen et al. [4] focus on similar comprehension challenges and also rely on collapse and expansion mechanisms to achieve scalability, though their visual abstractions are fundamentally different. In each case, a rendered view aggregates trace data using a hierarchical model of program entities (e.g., packages, classes, methods). Their “massive sequence view” portrays a snapshot of the global system trace as a time-ordered sequence of edges among...
higher-level program entities. Their “circular bundle view” represents calling patterns within a region of the global trace as slots around a circle. Each slot represents a system entity and the edges among them represent aggregated calls, with edge thickness denoting the cardinality of the aggregate. While the goals of our work are similar to those of Hamou-Lhdaj et al. and Cornelissen et al., there are important differences. Beyond the obvious language and platform distinctions, our work is focused on generating UML Sequence Diagrams. Given the common usage of this notation in many developer communities, our expectation is that the generated diagrams will be more amenable to adoption.

Orso et al. [29] describe an alternative approach to visualizing trace data collected from deployed software. Their focus is on scalable abstractions 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.

Reiss [34] describes an approach that abandons postmortem visualization in favor of realtime rendering. His approach, which relies on instrumentation to drive an attached visualization component, targets Java-based systems. The principal visual abstraction is the “box display”, where each box includes an enclosed rectangle used to represent the state of a given class or thread. Characteristics of the rectangle (e.g., height, width, hue, texture) are mapped to statistics concerning program entities. This includes the number of invocations on a given class, time spent in a particular thread state, number of synchronization events executed within the last epoch, and other statistics. The main display includes one box for every thread and class of interest. In [33], Reiss and Renieris extend these concepts to provide finer-grained detail at the level of basic blocks. More precisely, the authors describe an approach to relating thread activity to the basic blocks executed during each epoch. The visual abstraction, while more sophisticated, bears similarity to the box display. In this case, each box represents a single file and its internal elements represent basic blocks. Similar property mappings to those already discussed are used to represent the time spent in each block, object allocations performed by each thread, and other statistics. While the work of Reiss and Renieris has proven valuable in revealing high-level patterns of runtime behavior, their work is not well-suited to supporting comprehension activities associated with low-level control flow — our primary focus.

Closer to our work is that of Lange and Nakamura [18]. The authors describe a program visualization toolkit for C++ applications that combines static struc-
tural 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 recorder, 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 [22] describe a similar, but more advanced visualization system for C++ applications. The generated visual representations include class and method call graphs, UML communication diagrams, and UML sequence diagrams. The system additionally includes static and dynamic event filtering—which inspired aspects of our work—and supports dynamic visualization during the execution of a program under test. In contrast to these systems and others designed to reverse-engineer UML sequence diagrams [28], our work targets embedded network applications constructed using TinyOS [13, 20] (and nesC). 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.

2.2 Distributed Runtime Behavior

A range of efforts have focused on visualizing distributed behavior in the context of desktop- and server-class systems. CORBA systems have received particular attention. Moe and Carr [25], for example, describe an interceptor-based approach to collecting runtime performance data in CORBA systems. This data is aggregated in the form of scatter plots used to identify anomalous execution patterns. A plot comparing execution time across a series of calls could, for instance, be used to identify undesirable execution paths across distributed components. This approach, while well-suited to conveying aggregate data, is not designed to provide insights into low-level calling behaviors—the focus of our work.

Li [21] describes a similar instrumentation and visualization framework, but focuses on identifying causal connections among distributed events. Beyond standard probe injection, his approach relies on augmenting remote procedures with additional arguments used to track distributed call chains and thread ownership relations. Each injected probe records the relevant identifiers, call latency data, and statistics concerning resource usage (e.g., CPU utilization, memory usage). Upon termination, the collected data is used to recover the dynamic system call graph. Navigating the graph can reveal deep insights about the causal connections among threads, functions, latency, and resource consumption across a network. While the architecture shares similarities with our design, it does not consider local runtime behaviors (since the associated
calls do not pass through CORBA stubs). Further, in contrast to our work, the authors explicitly exclude consideration of asynchr

A number of visualization frameworks for parallel message-passing environments have been proposed. Nagel et al. [27], for instance, describe a suite of tools for visualizing applications implemented using the MPI model [38]. Trace data is collected for realtime analysis using the standard MPI profiling interface, as well as by custom probes specified by the developer. The generated views focus primarily on the percent of time spent at each participating node on a given set of tasks; both static snapshots and dynamic animations are provided. More interesting is the “timeline” view which depicts the message exchanges among MPI nodes. Similar to the work of Li, the system described by Nagel et al. does not provide detailed call sequence information for local processing tasks. The visual abstractions are also dissimilar to ours, which target UML users.

Wu et al. [44] similarly target applications executing on SMP clusters. Their visualization approach relies on “time-space plots” to summarize collected trace data over a specified time window. In this format, time is measured on the vertical axis; the units on the horizontal axis vary. Memory consumption might, for instance, be measured on this axis to depict the memory profiles of various processes over time. A significant contribution of their work is a clock synchronization technique used to correlate distributed events. As we will later see, our approach avoids the use of synchronized clocks (and timestamps) due to the severe resource constraints of the target platforms.

Topol et al. [41] present a similar, but more sophisticated visualization environment for cluster-based systems. A broad set of realtime and postmortem views are provided. Although there is support for depicting the message exchanges among nodes, the views focus primarily on resource consumption and other performance characteristics. By contrast, our work differs in its emphasis on source-level program understanding.

Closest to our work is that of Briand et al. [3]; the project goals are similar: Their objective is to generate UML sequence diagrams that capture the runtime behavior of distributed systems constructed using Java RMI. The approach is to project the global trace sequence collected during a run onto a single local sequence diagram. The name of each participating object is prefixed with an identifier that indicates its host address. Method invocations across distributed objects are then represented using standard sequence diagram notations. We follow a similar convention in our approach, but with several important differences: First, we provide interactive linking among causally connected events to reduce the cognitive burden associated

---

2 The authors present an excellent survey of similar tools.
with large datasets. Second, we automatically identify (and colocate) regions of logically concurrent activity across distributed participants. Finally, our system targets resource-constrained sensor nodes and captures activity with interrupt granularity.

In [5], we describe a program visualization testbed for sensor network systems. The work presented here leverages aspects of this system. We have not, however, previously considered local projections of distributed behavior, nor the automatic identification of logically concurrent activity regions. Consequently, the corresponding visual abstractions described here are new.

2.3 Sensor Network Development and Debugging

While program visualization techniques have received little attention in the sensor networks community, a number of efforts have focused on assisting sensor network developers. Several authors have, for example, described techniques for detecting and localizing software faults in sensor networks. Ramanathan et al. [30] 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. [15] 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 [15].

Improving runtime visibility has been another important focus in the literature. Tolle and Culler [39] 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. [43] extend this work; they describe a system for accessing program state without the need for manual attribute encoding. The toolkit

---

3 Two events $e_1$ and $e_2$ are logically concurrent if there is no causal connection between them. While $e_1$ and $e_2$ need not occur at the same instant in a given run, they could occur at the same instant in some run.

4 This work was only recently published; it leverages and extends the visualization system described in the original ICPC version of this manuscript.

5 This technique appears to represent a new software design pattern; it bears some similarity to the Flyweight pattern originally codified in [10].
additionally provides remote procedure call capabilities from a basestation to a network node. In [6, 7], we 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.

Finally, it is worth noting that a number of groups have recently released development environments for TinyOS [24, 35, 36]. 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.

3 Background: TinyOS and nesC

TinyOS applications are implemented in nesC [11, 19], 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\textsuperscript{6}, 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.

\textsuperscript{6} For the sake of presentation, we use the term component only in reference to nesC modules.
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 originating from the scheduler are said to execute in a synchronous context. By default, the scheduler executes tasks—which may invoke commands and signal events—in the order in which they were posted. Each task executes to completion before the next task is scheduled. Interrupts provide 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. To enable the compiler to distinguish between the two, commands and events callable from an asynchronous context include the async keyword as part of their signature. These asynchronous functions are prohibited from calling synchronous functions since the latter are not equipped to handle (logical) concurrency. We will see that this rule has important implications for the toolkit.

4 Visualization Toolkit

We now turn our attention to the design and implementation of the visualization toolkit. The steps involved in visualizing the behavior of an application are summarized in Figure 2. The first step is to analyze and transform the source base of the system under test. There are two outputs associated with this step. The first is an instantiated source base free of generic types. This instantiation process is critical to later visualization tasks; it enables individ-
ual component instances to be traced at runtime, improving the achievable visualization granularity. The second output is 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 logging probes that record enter and exit events on the functions of interest. If distributed behavior is targeted, radio transmission and reception functions are automatically selected for instrumentation.

The output of the selection step is an instrumented source base and a symbol map that enables numeric identifiers captured in a recorded trace to be associated with meaningful symbol names. The instrumented system is then compiled and executed on one or more hardware (or simulation) platforms. After execution, the resulting trace data is collected from each participating device. The data is then reconstructed using the symbol map to substitute symbol names for the numeric identifiers stored in the trace. If there are multiple participating devices, a message map is generated to map each recorded send() event to its corresponding receive() event. (This is achieved using message data introduced by the logging probes — a point we will return to later in the section.) Each reconstructed trace can be used to generate a corresponding sequence diagram that describes the local run. Alternatively, a subset of the traces may be used, along with the corresponding message map, to generate a sequence diagram that captures the system’s global behavior. The individual components of the toolkit that enable this process are described in the following subsections.
4.1 nesC Analysis and Instrumentation 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\(^8\). 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. We present an overview of the library’s functionality, referring the reader to [8] for a more complete treatment\(^9\).

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. 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 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.

---

\(^8\) The full nesC grammar is supported, as implemented by version 1.2.7a of the nesC compiler.

\(^9\) Again, this is a recent technical report authored after the publication of the original ICPC manuscript.
<table>
<thead>
<tr>
<th>Type</th>
<th>Method Name</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Traversal</strong></td>
<td>getNodesOfType(),</td>
</tr>
<tr>
<td></td>
<td>getParent(), ···</td>
</tr>
<tr>
<td><strong>Generation</strong></td>
<td>generateExprStmt(),</td>
</tr>
<tr>
<td></td>
<td>generateEnum(), ···</td>
</tr>
<tr>
<td><strong>Modification</strong></td>
<td>addComponentToConfig(),</td>
</tr>
<tr>
<td></td>
<td>instantiateGenerics(), ···</td>
</tr>
</tbody>
</table>

Table 1
A&I Library Methods (partial)

The final section lists two example code modification methods. The first method, 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 three 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
Listing 1. Instrumentation Procedure

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.

The final transformation is optional and only applied when tracing distributed behavior. First, as noted previously, the TinyOS functions used to transmit and receive radio messages are instrumented automatically. In addition, a custom version of the TinyOS radio stack is linked into the target application.
The modifications introduced in the stack are minimal: First, the radio message structure is augmented to include fields for the address of the sending node and a corresponding message sequence number. These values are added automatically to each message as part of the transmission process. At the point of reception, these values are stored as part of the corresponding log entry. Later, these values are used to map each transmission event to its corresponding reception event as part of the message map generation process.

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. Special commands are provided to log events associated with message transmission and reception. These commands are responsible for storing the source of the transmission and the associated message sequence number. Internally, the component implements a dual 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 flushed. 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 from participating nodes and reconstructed for use in generating the corresponding sequence diagrams. This is achieved by installing a new application image on each device. The installation process may be performed manually or through the use of an automation service. See, for example, the testbed-based approach described in [5]. In either case, the Trace Extractor application 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. Finally, if the user is interested in visualizing distributed behavior,
the Trace Collector is responsible for producing the message map used to associate transmission and reception events. Both the reconstructed traces and the message map are saved for later use in generating sequence diagrams.

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 Local Sequence Diagram Generator

The Local Sequence Diagram Generator is used to transform a reconstructed trace into a sequence diagram that captures the associated local behavior. 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 coloring scheme summarized in Table 2. 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
<table>
<thead>
<tr>
<th>Type</th>
<th>Color</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calls / Returns</td>
<td>Black</td>
<td>Synchronous</td>
</tr>
<tr>
<td></td>
<td>Red</td>
<td>Asynchronous</td>
</tr>
<tr>
<td></td>
<td>Blue</td>
<td>Module functions</td>
</tr>
<tr>
<td>Activations</td>
<td>Orange</td>
<td>Commands</td>
</tr>
<tr>
<td></td>
<td>Purple</td>
<td>Events</td>
</tr>
<tr>
<td></td>
<td>Blue</td>
<td>Tasks</td>
</tr>
<tr>
<td></td>
<td>Green</td>
<td>Module functions</td>
</tr>
</tbody>
</table>

Table 2
Sequence Diagram Edge Coloring Scheme

text 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.

4.7 Distributed Sequence Diagram Generator

The design of the Distributed Sequence Diagram Generator parallels that of the local version, but includes support for representing behaviors that span multiple input traces. The visualization approach is to project distributed components onto a single local diagram. The generated diagram includes a component rectangle for each of the participating component instances. In each case, the component name is prefixed with the address of the hosting device. Behaviors local to a host are represented using the same conventions described above.

Three enhancements are applied to depict distributed interactions and to assist developers in reasoning about the causal connections among events. The first and most basic of these enhancements is the addition of special markers on the activation bars of relevant components to depict transmission and reception events. A transmission event is depicted as a small closed-envelope icon. Reception events are similarly depicted, but use open-envelope icons.

The second enhancement supports interactive exploration of the message exchanges among nodes. When a user clicks a transmission icon, a small popup list of the associated recipients is displayed. The display does not disambiguate unicast versus broadcast; however, only those recipients that actually received a given message are displayed. This provides immediate feedback regarding the impact of packet loss — a common occurrence in low-power wireless networks. Further, by selecting one of the recipients, an orange line is added to
the display to connect the transmission event to the corresponding reception event. Traversing this line leads to the corresponding point on the recipient's activation rectangle. Additional transmission lines can be added to the display using the same process and a simple clear button is used to remove the lines if the display becomes cumbersome. In this way, developers can explore selected interaction sequences across a network without being overwhelmed by all of the interaction sequences.

The final enhancement is perhaps the most interesting. The objective is to highlight distributed action sequences that are logically concurrent. We adopt the definition of logical concurrency based on the happens before relation, as developed by Lamport [17]. The relation is intended to represent causal connections among events. In short, an event $e_1$ is said to happen before $e_2$ if $e_1$ precedes $e_2$ in the same local trace, or $e_1$ and $e_2$ are corresponding transmission and reception events; the relation is transitive. Two events are considered to be logically concurrent if there is no causal connection between them. Or more formally, events $e_1$ and $e_2$ are logically concurrent if neither event happened before the other. Highlighting logical concurrency is important for two reasons. First, it limits the set of interaction paths that must be considered to reason faithfully about a distributed computation. Second, if two events are logically concurrent, they need not occur at the same instant in the given run — but they could occur at the same instant in some future run.

The visualization approach is based on a partitioning strategy that collocates, along the vertical axis, activity sequences that are logically concurrent. Collocating these sequences along the same segment of the timeline emphasizes that they may have occurred at the same instant. Consider a computation involving three nodes. Upon termination, three trace sequences will be collected, $\tau_1$, $\tau_2$, and $\tau_3$. The partitioning process segments each trace into an equal number of subsequences such that $^{11}$:

\[
\begin{align*}
\tau_1 &= \tau_{11} \ast \tau_{12} \ast \ldots \ast \tau_{1n} \\
\tau_2 &= \tau_{21} \ast \tau_{22} \ast \ldots \ast \tau_{2n} \\
\tau_3 &= \tau_{31} \ast \tau_{32} \ast \ldots \ast \tau_{3n}
\end{align*}
\]

The sequence partitioning is subject to two key requirements. First, for each subsequence $\tau_{ij}$, every constituent action must be logically concurrent with respect to the actions contained in the corresponding subsequences $\tau_{kj}$ for each node $k$. That is, the set of subsequences at position $j$ must be logically concurrent subsequences $^{12}$. Second, the subsequences must be maximal in the sense that no additional actions could be included while preserving the concur-

---

$^{11}$ We use the $\ast$ operator to denote concatenation.

$^{12}$ Two subsequences $\tau_{ij}$ and $\tau_{kj}$ are logically concurrent if and only if there are no transmission events in $\tau_{ij}$ with corresponding receive events in $\tau_{kj}$.
rency requirement. In this way, the longest subsequences of activity that are logically concurrent are batched at the same position. When rendering the corresponding diagram, all subsequences at position $j$ span the same vertical space on the timeline. Horizontal bars are also included to demarcate subsequence regions. In this way, if two actions are on either side of a horizontal bar, there may be a causal connection. If they are within the same bounded region, there is no causal connection between them. We will see that this approach highlights interesting distributed phenomenon in a straightforward manner.

4.7.1 Callers, Callees, and Visualization Uncertainty

It is useful to consider the rules used to associate callers and callees, and to describe an interesting limitation that arises as a result of these rules. Specifically, the rules inject a degree of uncertainty in the generated sequence diagrams: There are cases in which a sequence diagram may not precisely reflect a system’s runtime behavior. To understand the source of this uncertainty and to gauge its impact on the usability of the toolkit, it is best to consider an example.

Consider a program $P$, for which some subset of the program actions will be instrumented and traced. We refer to this set as the instrumentation set of $P$. A portion of the call graph for $P$ is shown in Figure 3. Each circle represents a program action; shaded circles denote actions in the instrumentation set. Without a loss of generality, assume that $S()$ is the closest instrumented action between $S()$ and $D()$.

Now consider generating a sequence diagram based on a trace of $P$. In particular, consider an element of the trace, $D_b$, corresponding to an enter event on action $D()$. The basic visualization task is to determine which action invoked $D()$ — either directly, or through a call chain outside the instrumentation set.

---

13 Subsequences at position $j$ need not be of uniform length. Indeed, some subsequences may be empty.
of P. It is important to observe that a simple projection of the runtime stack at $D_b$ can be constructed by tracing the enter and exit events recorded on the trace. The projection includes (conceptual) activation records only for those actions in the instrumentation set.

There are two cases to consider. First assume that $D()$ is a synchronous action. In this case, the visualization assigns the topmost synchronous action on the runtime stack at $D_b$ as $D()$’s caller. Assume that this action is $S()$. To see why this assignment is correct, recall that TinyOS prohibits asynchronous actions from invoking synchronous actions. Hence, any call chain flowing through $D()$ must have originated from a synchronous context. Similarly, since $S()$ may invoke $D()$ (according to the system call graph), $S()$ must also be synchronous, and the activation of $S()$ on the runtime stack must have originated from a synchronous context. Since a synchronous flow cannot preempt another synchronous flow, the call chain leading to $D()$ must have come from $S()$. If $S()$ were not active (i.e., there were no synchronous actions on the runtime stack at $D_b$) the call chain that lead to $D()$ did not pass through any of the actions in the instrumentation set. In this case, the generic System action is assigned as the caller.

Now assume that $D()$ is asynchronous. In this case, the path to $D()$ may have come from either a synchronous or asynchronous context. In assigning a caller to $D()$, the system again examines the runtime stack at $D_b$, excluding actions that could not have lead to $D()$. If there are no suitable candidates, the System action is assigned as $D()$’s caller. Otherwise, the topmost candidate —synchronous or asynchronous— is selected; assume that this action is $S()$. In some cases, this assignment is incorrect.

Under what conditions could $S()$ be the topmost action on the runtime stack (at $D_b$) according to the captured trace, but not be responsible for the call to $D()$? There is only one possibility: The execution path through $S()$ must have been interrupted by a new call chain that reached $D()$. This chain must satisfy two key properties. First, it must have originated from an asynchronous context since synchronous paths are non-preemptive. Second, it must flow along an uninstrumented path in $P$; otherwise $S()$ would not be the topmost candidate recorded on the captured trace. Again consider Figure 3. An asynchronous flow through $Y()$ to $D()$ could interrupt a call chain from $S()$ to $D()$. The visualization system would incorrectly associate the first call to $D()$ with $S()$, and the second call to $D()$ with $Y()$. Note that this association error would not occur if a preemptive path through $X()$ reached $D()$ since $X()$ is included in the instrumentation set of $P$. Hence, in general, an association error will occur if an asynchronous flow reaches an asynchronous action that is reachable from

---

14 We include module-level functions in this set since they may be invoked from an asynchronous context.
The frequency of association errors depends on the application under test, as well as the corresponding instrumentation set. In practice, however, the occurrence rate seems likely to be rare. The risk can also be reduced by introducing additional instrumentation points to disambiguate asynchronous call chains. Still, a limited degree of uncertainty exists, and it would be preferable to reduce or eliminate it. We will return to this point in Section 7 when considering future research directions.

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. A third example, BarrierSync, centers on a custom application involving distributed coordination. As we will see, 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 4 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 5, 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 Local Sequence Diagram Generator to produce the diagram shown in Figure 6.

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 TimerFrom.fired() on VirtualizeTimerC, which in turn invokes a local dispatching function, fire-

---

Fig. 5. Probe Selector

---

15 The 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.
Fig. 6. Sequence Diagram for Blink

Timers(). 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 sec-
ond 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 7.

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\textsuperscript{16}.

5.3 \textit{Scenario 3: BarrierSync}

The third scenario centers on \textit{BarrierSync}, a custom application that implements a standard distributed coordination protocol — namely, \textit{barrier synchronization}. Each of the participating nodes periodically blocks at a \textit{barrier point}. The critical protocol requirement is that each node must wait at the barrier until all other nodes have reached the same point. The implementation relies on a simple messaging strategy where each node broadcasts its arrival at the barrier and blocks until it has heard from all other network nodes. This

\textsuperscript{16}We were unaware of the automatic address filtering performed by the CC2420. This behavior is immediately obvious from the sequence diagram generated when \textit{address recognition} is enabled on the radio chip. This feature was disabled to generate the above diagram.
process then repeats. For the sake of presentation, the protocol is implemented using a reliable messaging layer that ensures message delivery. The test system consists of five participating nodes, but only three are included in the visualization. We again omit the individual visualization steps and skip to the generated sequence diagram shown in Figure 8.

The component labels at the top of the diagram indicate that the diagram captures activity across three nodes (i.e., Mote 0, Mote 1, and Mote 2). The blue horizontal lines at the top and bottom of the figure denote a subsequence break. Hence, the activity sequences at each node are logically concurrent. Or stated another way, there are no causal connections among the participating nodes at this point in the execution sequence. (Only a small portion of the execution sequence is displayed.) This pattern of activity is repeated (with some variation) for the lifetime of the network.

A more detailed examination of the display reveals that each subsequence in the diagram begins with the receipt of a message, as indicated by the open-envelope icons. This matches our intuition since the reception of a message demarcates a causal connection between two nodes. When this barrier arrival message is received, the Receive.receive() event is signaled on BarrierSyncP, which in turn updates its message tally and invokes the private function checkAndFireTimer(). This function is responsible for determining whether a node can cross the barrier. At the first node, this function immediately returns true, indicating that the last message received completed the tally; the node may cross the barrier. In response, Receive.receive() starts a new timer to simulate processing before the node again reaches the barrier, as signaled by Timer.fired(). At this point, the node transmits barrier arrival messages (in sequence) to each of its neighbors, as indicated by the closed-envelope icons. It then repeats the process of waiting to complete its tally.

The subsequences corresponding to the second two nodes are similar. The only substantive difference is that each node requires two barrier arrival messages before passing the barrier. The remaining details are clear from the figure. This basic pattern of activity repeats indefinitely. The only variation is in the placement of the subsequence breaks, which change in each round due to the non-determinism associated with message transmission and delivery.

We note that orange lines are added to the display to connect corresponding transmission and reception events whenever the user clicks an envelope icon. The clear button at the bottom of the frame is used to remove these lines if the display becomes too cumbersome. We omit an additional screenshot for the sake of brevity.

While the scenario is focused on a simple application, it illustrates the potential of the toolkit: The local projection approach provides a straightforward
Fig. 8. Sequence Diagram for BarrierSync
view of the network’s global behavior. Demarcating regions of logical concurrency further clarifies network behavior, drawing attention to key system execution points. Finally, the icon-based traversal strategy makes it easy for developers to trace interaction sequences that span distributed nodes and to understand the effects of packet transmission and loss.

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 and records the duration of each call. The process is repeated for three different event record types ranging in size from 1 byte to 4 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 3. 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 buffer capacity was set to 70 records for cases 1 to 3, and 50 for case 4.) 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,

---

17 Recall that the Event Recording Service is implemented as a thin wrapper over the LogStorageC component provided by TinyOS. Given a fixed event record size, assessing the maximum capture rate is equivalent to evaluating the performance of LogStorageC for the target hardware (or simulation) platform.

18 The variation is likely due to the underlying buffering strategy implemented by LogStorageC.
Table 3  
Event Logging Time

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Buffer Count</th>
<th>Time (seconds)</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3448</td>
<td>0.008</td>
<td>1.051</td>
<td>0.009</td>
<td>0.032</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1856</td>
<td>0.011</td>
<td>1.054</td>
<td>0.014</td>
<td>0.045</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1216</td>
<td>0.014</td>
<td>1.058</td>
<td>0.185</td>
<td>0.058</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1280</td>
<td>0.014</td>
<td>1.017</td>
<td>0.018</td>
<td>0.052</td>
<td></td>
</tr>
</tbody>
</table>

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 4. The *Baseline* columns correspond to the first configuration; the *EEPROM Logging*
columns correspond to the second configuration; and the Full Instrumentation columns correspond to the third configuration. The RAM results are summarized in Figure 9, and the ROM results are summarized in Figure 10. 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 a 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 11. 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.

Table 4
Base Overhead (RAM/ROM)

7 Conclusion

Our work began with an interesting observation. The efficiency characteristics that drive the continued adoption of TinyOS [13,20] 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-efficient systems, the model introduces non-determinism in the execution ordering of program actions. This issue is exacerbated by the application contexts in which TinyOS is applied — embedded network systems are inherently parallel and distributed. 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, especially in the presence of distribution. 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 latter diagrams come in two variants: The first follows standard UML conventions and depicts local runtime behavior. The second introduces several enhancements to capture global runtime behavior across distributed participants. 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 across a network, reconstructing the associated runtime traces, and correlating transmission and reception events. Finally, we described three visualization front-ends corresponding to the afore-

Fig. 11. Incremental Overhead (ROM)
mentioned views. The application of the toolkit was demonstrated using system examples included as part of the standard TinyOS 2.0 distribution, as well as a custom example involving distributed coordination. A detailed analysis of the toolkit’s resource usage and performance characteristics was also presented.

Looking forward, we plan to pursue two avenues of exploration. First, we plan to investigate techniques for reducing the degree of uncertainty associated with the visualization of asynchronous flows. While additional probes and/or instrumentation logic could be used to reduce (or even eliminate) this uncertainty, resource limitations reduce the set of applicable solutions. One approach that we plan to investigate is based on context-sensitive static analysis, as in [23,32]. The central idea is to exclude the consideration of flows that originate from a disabled interrupt context when determining whether an asynchronous flow was injected between two logged events. Second, we plan to investigate new visualization techniques for embedded networks based on the application of the nesC Analysis and Instrumentation Library.

A preliminary version of this work [9] was published in the proceedings of the 16th IEEE International Conference on Program Comprehension. The manuscript also leverages work reported in [8] and significantly extends work reported in [5].

8 Acknowledgments

This work was supported in part by the National Science Foundation through awards CNS[CAREER]-0745846, CNS[NeTS-NOSS]-0520222, and CNS[ITR]-0218495. The authors gratefully acknowledge the NSF for its support.

References


