12-2009

An Animation Framework for Improving the Comprehension of TinyOS Programs

Sravanthi Dandamudi
Clemson University, sdandam@g.clemson.edu

Recommended Citation

Dandamudi, Sravanthi, "An Animation Framework for Improving the Comprehension of TinyOS Programs" (2009). *All Theses*. 736.
https://tigerprints.clemson.edu/all_theses/736

AN ANIMATION FRAMEWORK FOR IMPROVING THE COMPREHENSION OF TINYOS PROGRAMS

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Masters of Science
Computer Science

---

by
Sravanthi Dandamudi
December 2009

Accepted by:
Dr. Jason O. Hallstrom, Committee Chair
Dr. John McGregor
Dr. Brian Malloy

Abstract

To meet the increasing demand for monitoring of the physical world, there has been an increase in the development of wireless sensor network applications. The *TinyOS* platform has emerged as a de facto standard for developing these applications. The platform offers a number of advantages, with its support for concurrency, power-efficient operation, and resource-constrained hardware chief among them. However, the benefits come at a price. Even without the TinyOS platform, the inherent parallel and distributed nature of these applications makes it difficult for developers to reason about program behavior. Further, the TinyOS programming model adopts asynchronous, split-phase execution semantics. Developers must explicitly manage program control state across event-handlers, components, and devices. This makes the design, debugging, and comprehension of these programs even more difficult.

In this work, we describe an animation framework for TinyOS programs, designed to enhance the comprehension of their runtime behavior. The framework enables application developers to specify, in the form of an XML configuration file, the runtime elements to be captured within a given system and the manner in which those elements should be displayed. The resulting visualization presents an animated play-back sequence of the events that occurred during execution. The framework also provides a visual representation that connects causally-related events in a distributed network. We describe the design and implementation of the animation framework and present an analysis of the runtime overhead it introduces.

Dedication

For *Amma* and *Nana*.

Acknowledgments

I owe my deepest gratitude and respect to my advisor, Dr. Jason O. Hallstrom. If it were not for his ideas, support, guidance and motivation, none of this would have been possible. I thank him for making a difference in my career. I would like to thank the faculty of Computer Science and Information Systems group at BITS, Pilani, India and the faculty at the School of Computing, Clemson University for laying the required foundations.

I also thank Dr. Andy Dalton and Sally Wahba for their encouragement and suggestions. I am greatly in debt to Karthik and Suman for their amazing company. A special thanks to everyone who wished good for me. Last, but not the least, I thank my parents for all their love and sacrifices and for standing by me for my aspirations.

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

Wireless Sensor Networks observe the physical world to provide useful information otherwise unavailable to humans. They are usually deployed over large geographical areas or in hostile environments. These networks are composed of tiny computing devices that sample data from onboard sensors, process that data, and communicate the results to peers. Applications of these networks span multiple domains. Examples include systems developed for environmental monitoring [4], volcano monitoring [14], structural monitoring [9], early warning of natural disasters [3], building-scale power management [21], intrusion detection [1], and more recently, social networking [26].

These applications present a unique set of requirements, and the hardware platforms that host them present a unique set of constraints. Specifically, the applications require a high-degree of distribution and reactivity, while the hardware platforms afford orders of magnitude less resource than standard desktop machines. The applications run for unusually long durations in harsh environmental conditions on devices with limited computational capacity, memory, and bandwidth.

1.1 Motivation

To address the challenges posed by sensor network applications, the *TinyOS* development platform has emerged [19]. The platform provides an event-driven programming model and supporting libraries, which together enable the construction of lightweight, power-efficient applications that accommodate a high degree of concurrency. It has emerged as a de facto standard.

However, wireless sensor applications are difficult to design and debug. The difficulty stems from the inherent distributed behavior of the applications and the programming-model adopted by TinyOS. Developers must manage an application consisting of sensor nodes deployed in a distributed environment. Unreliable connectivity between nodes, caused by either node failures or signal issues, introduces non-determinism in the execution of these applications.

The programming-model of TinyOS is based on asynchronous, split-phase operations. The operations are driven by hardware interrupts that react to external events, such as completing a sensor acquisition/request, or receiving a message from a neighboring node. Rather than providing blocking semantics, the programming model is based on a non-blocking call mechanism, where requests made to components to execute an operation return immediately; events are signaled upon completion. Application logic is distributed across event-handlers, which are themselves distributed across compilation units. The motivation for this design choice is simple: When executing long-running hardware operations (e.g., radio transmissions, flash storage updates), blocking semantics are typically provided through either a threading system or a busy-wait mechanism. The former requires additional memory resources and the latter requires additional processing (and hence, energy) resources. Both are incompatible with target hardware platforms.

1.2 Problem Statement

The efficiency benefits afforded by TinyOS come at a significant expense. Program understandability becomes difficult when developers must handle the inherent distributed nature of wireless applications and must design systems which adhere to a programming model that introduces non-determinism. They are tasked with explicit management of the control state that governs the behavior of event handlers—thus hindering their ability to understand execution behavior.

1.3 Solution Approach

To improve program comprehension, we present a program visualization framework designed to aid in understanding TinyOS program behavior. The framework supports customizable animations that clarify the runtime interactions among application components and the impact of those interactions on program state. The framework relies on an XML-based visualization language used to specify the program elements of interest and the display properties that should be used to render those elements. For a given application, the corresponding visualization document governs the behavior of an instrumentation engine that injects logging probes at appropriate execution points to collect a runtime trace. Later, the collected trace data is extracted from the embedded host and replayed through a visualization front-end that mimics the appearance of a standard media player, providing a movie-based abstraction of the recorded trace. The framework also facilitates tracking message exchanges between nodes and shows causal relations between events in a distributed network. We describe the design and implementation of the visualization framework in the context of a running system example and present an analysis of the runtime overhead it introduces in instrumented applications.

1.4 Contributions

The contributions of this thesis are as follows: First, we present a configurable animation system to help in comprehending local program behavior of embedded wireless systems. Second, we extend the local visualization view to correlate messages exchanged between the wireless devices to aid in understanding distributed behavior of causally-related events. Finally, we evaluate the framework in terms of memory usage and number of events captured.

1.5 Thesis Organization

Chapter 2 surveys some of the most closely related work in program visualization. Chapter 3 provides a brief overview of the TinyOS programming model. Chapter 4 presents the design and implementation of the animation framework. Chapter 5 discusses use-case scenarios to demonstrate the utility of the animation framework. Chapter 6 describes the results of our performance analysis. Finally, Chapter 7 presents a summary of conclusions.

Chapter 2

Related Work

In the domain of desktop software, program visualization has a long history of exploration. In this chapter, we present some of the most relevant related work. The next section presents visualization techniques for understanding the *structure* of software systems. In section 2.2, related work focused on *behavioral* visualization is discussed. Section 2.3 surveys some *animation* systems. Finally, section 2.4 discusses visualization work in *embedded systems*.

2.1 Structural Visualization

Ball and Eick [2] present visualization techniques that aid in understanding the structure of large systems (i.e., greater than one million lines of code). The authors claim that the usual graphical tools used for system visualization cater only to small systems and require a deep understanding of program code. Their tool allows a user to choose either line, pixel, file summary, or hierarchical representations for visualizing program text, text properties, and file statistics. The rendered display is color-coded based on the property of interest. Filtering capabilities provided by the tool allow users to view specific details of interest. For example, a user can deactivate certain regions of code, or can turn-on specific colors to show differences in code. The tool uses these representations to track code changes and maintain version history, display static properties of programs, and discover program *"hot spots"*. The tool is built using a cross-platform C++ library. The tool does not, however, visualize the runtime behavior of applications and focus mainly on program text visualization.

Brade et al. [6] describe a tool to aid in software maintenance by providing hyperlinks between the program text of various components interacting in a software system. Multiple views display various aspects of a program simultaneously. For example, one view shows all the variables used in a system, while another shows a graphical representation of code fragments that are conceptually related, but located at different physical locations (i.e., in different components). The graphical representation shows a static call graph based on functions identified by a user. A function selected by the user is highlighted in the displayed views by assigning a color to each function. Users can also inquire about information pertaining to variables — specifically the functions where they are modified. Their tool is useful in understanding the structure of a software system.

Orso et al. [29, 30] present a tool to visualize both structure and behavior. An instrumentation technique is used to collect execution data related to performance, CPU usage, etc. A user can view program code at different levels of detail by using various visual representations supported by the tool. Different levels of detail relating to program code is displayed to the user. For example, a statement-level view presents the source code using an appropriate coloring scheme for each line of text. The file-level view also uses the coloring scheme to show a condensed view of the source code. It uses a technique introduced by Eric et al. [20] to map each line in the source code to a row of pixels. In a system-level view, a tree-map representation shows the system structure. This technique was developed by [36]. Each node in the tree corresponds to a class or a package, represented as a rectangle whose area is proportional to the size of the class/package with respect to the complete system. In addition, *execution bars* represent multiple executions of a program. Each vertical band in a horizontal bar corresponds to data collected during a given run of the program. Filtering and summarization capabilities are supported to allow users to select a subset of the executions for visualization. The tool does not, however, aid in reasoning about program control flow, nor does it provide insight about program state, which is the focus of our work.

2.2 Behavioral Visualization

When considering visualization of runtime behavior, the relevant literature is extensive (e.g., [22, 5, 24, 32, 10]). Bohnet and Döllner [5] present a tool for understanding the functionality of large software applications implemented in C and C++. The tool facilitates locating software components responsible for implementing particular *features* (a particular functionality provided by the software system). It combines static information extracted from a systems' source with runtime trace data to generate dynamic call graphs. The trace collection system avoids the need to instrument source code. Instead, the program under study is executed within a wrapper process that logs data used for analyzing the applications' behavior. The user interacts with the system to execute a particular functionality of the system. During this process, system trace data is logged. This information is then combined with static information detailing the relationships between various classes. Functions that are responsible for the implementation of the executed feature are shown as a call graph. The user can focus on a particular function, which is then represented as a disc; arrows tops and arrow bottoms represent predecessor and successor functions (of the focused function), respectively. Calls between functions are represented by arcs. Also, the source code of the selected function is shown below the graphical view. The visualization front-end provides navigation facilities that allow the users to explore specific regions of the call graph. Both the graphical and the source code view presented by the tool assist the user in exploring a given system feature. Their tool analyzes only a specific execution path for an identified feature since the path is based on a sequence of user interactions. Based on the visual presentations, the user can decide if a function is responsible for a feature implementation.

Cornelissen et al. [10] describe a similar trace comprehension tool to locate, explore, and comprehend features in software. Their approach relies on an interactive and scalable visualization mechanism to present the structure of system entities and the relationships between them in the form of *circular bundle views*. The *massive sequence views* present a complete overview of a global execution trace. A circular bundle view corresponds to a snapshot of an execution trace at a particular instance, where system entities are shown as slots around a circle. Edges between the slots are shown as *bundled splines* to represent relations between the systems entities. The number of calls between two entities is shown by the thickness of the splines. The tool also uses colors to distinguish calls based on their occurrence. The massive sequence view orders call relations along a vertical time-axis and shows the system structure along the horizontal axis. As before, calls are color-coded. The visual representation in our work mimics a movie player to show an animated view of events occurring during execution and the presented animation helps in understanding program control flow in a TinyOS application.

Reiss [33,17] presents a visualization system focused on improving developers' understanding of the execution of classes and threads in JAVA programs. Each class or thread is shown as a rectangular box and information related to these objects is captured in one or more colored boxes residing inside the outer box. Each statistic (number of entries in a class or number of allocations etc.) is displayed using different horizontal and vertical sizes of the box or by using different colors. The height of the inner rectangle, for example, corresponds to number of calls and the width corresponds to the number of allocations performed by methods of the class. To reduce visual clutter, the tool divides a system execution into distinct intervals and displays an execution summary for each interval. An XML file is used to identify all routines from the JAVA libraries that affect thread state. Entry and exit events of these routines are then used to track the state of the thread. Also, information collected during execution is saved in an XML document and passed on to the visualizer. The author extends this work to provide more detailed information about program execution [31]. The visual representation differs in this case. Each box represents a source file, and sub-divisions in the box correspond to thread-related and basic block information. Statistics related to program properties such as number of instructions executed, number of threads currently executing, etc., are represented as different colors inside the inner-most rectangular boxes. Theses statistics help in identifying performance bottlenecks. The work is further extended to provide fine-grained information about program behavior [32]. AspectJ is used to weave instrumentation calls into the program to automatically identify events that describe program behavior. These include function calls, returns, exception handlers etc. Each of these events are parameterized with program data (e.g., a method entry event can take as parameter a data object that is accessible from the method). Based on these events, an extended finite state automata representing program behavior is constructed. The automata describes the set of states in the execution and also provides insight about state transitions. The automata is then used for presenting the visualizations in the form of line plots, dot plots, or time bars. Each of these visual representations show different aspects of the program. Dot plots, for example, show dependencies between different methods, while time bars show program state at different instances of time. The description of the visualization is written to an XML file so that it can be recreated. The XML file contains the program elements, automata representing program behavior, the constructed data structures, and also the graphical properties of the visualization.

Sneed [38, 37] presents a tool which also relies on static and dynamic analysis techniques to aid in comprehension of object-oriented systems. Static information about the statements and functions in an application is maintained in *statement* and *function tables*, respectively. A function table contains a list of function names and their associated source information, i.e., the name of the source file containing the function. A statement table stores information about program statements and their corresponding line numbers. A user steps through each statement of program code and dynamic analysis occurs at every step. The dynamic analyzer refers to the statement and function tables to produce sequence diagrams. This tool also facilitates instrumenting source code with probes that collectively record the execution path. The recorded execution sequence can be used to draw sequence diagrams without the need for user to step through each statement. This saves time for a user who wishes to consecutively visualize the same execution path. Contrary to the above work, which targets object-oriented programs, our work specifically focuses on visualizing the runtime behavior of event-driven applications that run on memory-constrained devices.

Lange and Nakamura [22] describe a visualization tool for C++ programs that leverages both static and dynamic information. The tool provides mechanisms for collecting runtime trace data from an application and presenting fine-grained views of its behavior. In particular, their tool displays the interactions between classes and between objects to aid in understanding application structure and behavior. Their architecture is similar to ours; it comprises an instrumentation system, a trace recorder, and a rendering front-end. In addition, their tool provides analysis services that collect structural information about an instrumented program, including its constituent inheritance and containment relationships. This information is used in concert with the recorded execution data to present *object-centric* views of the application's behavior.

Malloy and Power [24] describe a more sophisticated tool for visualizing the behavior of C++ programs. Their *Selector* enables static selection of the classes and methods to be visualized at runtime. The tool then presents an overview of the program in the form of a class diagram and shows call graphs to aid in comprehending class relations and method interactions. Statically selected classes and methods are instrumented with profiling code. A *Profiler* interacts with the instrumented system to provide detailed runtime information about program interactions in the form of sequence and communication diagrams.

Moe and Carr [27] describe a trace-based approach to monitoring and configuring distributed systems. Runtime information is collected using *CORBA interceptors* and sent to a remote server. The raw data from different hosts is processed to construct a logical trace by eliminating local times of each host and mapping to a global time reference. Call sequences are built automatically. Summary statistics, including response time, number of calls, and other figures, are also calculated. A two- or three-dimensional space is used to present these statistics in the form of scatter plots.

The statistical parameters can be configured for presentation purposes. The user can choose to represent each statistic in a different colors or shapes. Dynamic filters allow querying values related to a particular statistic. These plots can be analyzed to discover anomalous behavior. For instance, the tool was used to identify errors in a server initialization by visualizing the call statistics of calls originating from the server component.

Briand et al. [7] also describe a visualization approach to assist in comprehending execution traces from object-oriented systems. Information collected during dynamic analysis is used to produce UML sequence diagrams. The program under test is executed multiple times to invoke all possible executions of a system use-case. Their approach is similar to ours; source code is instrumented, and distributed trace data is collected at runtime. This data is then used to present a single sequence diagram that shows method invocations. The approach first identifies an instrumentation mechanism to effectively represent the execution traces in the form of a sequence diagram. For this purpose, a trace meta model is constructed. The extracted trace data is an instance of the trace meta model. The tool relies on a set of consistency rules to relate trace data to the sequence diagram being produced. To distinguish objects from various hosts, objects in the sequence diagram are tagged with the host address. The tool does not, however, aid in identifying the causal relations among events in a distributed environment. Our work focuses on visualizing distributed program behavior on memory-constrained devices and presents visualizations in an animated way rather than in the form of a sequence diagram. Also, our work helps in understanding the changes in program state with respect to variable values.

Li [23] presents a framework for monitoring system behavior in large embedded applications. A source-based instrumentation strategy is used to insert probes. These probes track data that is later used to summarize system behavior. Data is primarily used to understand caller/callee relations. Causal events are traced and dynamically linked to provide a comprehensive view of system-wide behavior. The data is collected in a hierarchical manner. First, each process maintains an in-memory buffer to capture logged data records. Second, the buffer is saved on a local file system whenever it is full. Logged data scattered across a distributed system is collected and stored in a central repository. Their system supports applications running on multiple processors. Since the in-memory data is machine-dependent, the data is transformed to a machine-independent format. During the analysis phase, the reconstructed data, a dynamic call graph spanning across distributed components is displayed. The call graph additionally captures causal relations between threads of the system. The framework also provides mechanisms to selectively choose system components for visualization, either at compile-time or runtime.

2.3 Animation Systems

Sajaniemi and Kuittinen [34] present an educational tool to help novice programmers understand high-level programming constructs. The tool focuses on understanding the values of variables during program execution. The authors use images to represent variable values and to provide insight into the type of a variable; values that do not change after initialization, for example, are shown engraved on a stone. The authors describe a *role* of a variable as the nature exhibited by the variable during execution. For example, a variable whose value does not change during program execution is given a *fixed value* role, while a variable containing the sum total of values is said to have a *gatherer* role. Images are animated to show a change in the role of a variable. Additionally, color-coding enables users to distinguish between current and previous values of variables. Animation commands are inserted into actions that cause a change in the value or role of a variable. These commands record all variable changes. Animation commands must be inserted manually. Programs written in any programming language, such as C, JAVA, or Pascal can be animated by introducing appropriate animation commands. Their system does not recognize the roles of variables automatically and requires manual instrumentation for every program. In our work, we also support user-defined images to show function invocations, returns, and changes in program state. Our work helps TinyOS developers understand program behavior during execution and is not focused on teaching programming-constructs to beginners.

Brown and Sedgewick [8] describe an educational tool to aid students in understanding algorithm implementations in Pascal. A visual representation of the data structures used in an implementation and the values they hold are displayed. The system presents a *"motion picture"* of the algorithm in execution. Properties that cause a change in display are identified, and the information is passed to a *graphics designer* which drives the animation of the algorithm. The graphics designer generates *views* when the algorithm executes and updates the display to reflect the changes. The system also supports viewing past events by maintaining a history of execution. Users can open views even if the algorithm is already executing. The view updates itself to the current state of the algorithm instead of playing the complete execution from the start. However, a major drawback of this system is that every algorithm needs to be manually prepared for animation by identifying the events of interest.

Stasko [39, 40] describes another animation framework for visualizing algorithms. The tool provides mechanisms to represent various attributes of an algorithm by creating graphical objects such as rectangles, circles, etc. Smooth movements of the graphical objects show the execution of an algorithm. For example, while animating a binary search algorithm, a rectangle represents each element of the binary search tree and a bouncing circle is shown on an element while the search is being performed. The circle stops bouncing when the desired element is found. For producing the animation, the tool uses a trace file containing the mapping between program actions and the graphical objects used for rendering the display. Then, program code is annotated with calls that are dispatched to a central message server using the UNIX inter-process communication mechanism during execution. The central server then passes these messages to animation routines in C, which create the required graphical objects rendered on screen. Messages are stored in a central server to enable replay of the animation. This system was primarily developed to create animations in a relatively short amount of time compared to the application developed by Brown and Sedgewick [8]. Both these systems were designed to aid in understanding how *algorithms* perform, and do not focus on understanding software systems in general.

2.4 Embedded System Visualization

Gracioli and Fischmeister [18] describe a tracing mechanism designed to support improved understanding of interrupt behavior in embedded systems. When an interrupt occurs, a frequency-based *selector function* selects elements of the execution context, consisting of I/O registers, control registers, thread control blocks, etc. A hash-key is computed based on these selected components. Next, the return address from the execution context is extracted and a reduction function is applied to condense its bit representation. The hash-key and the reduced return address comprise a *fingerprint* which is stored in a database. To replay the application, a simulator computes a fingerprint after every instruction in the application and checks the database to determine if an interrupt occurred at a particular instruction. The motivation for this work is similar to ours — understanding complex control flow in embedded applications. However, the accuracy of their system is based on the selector and reduction functions used to compute the fingerprints and can therefore ignore some control flow paths. Our work provides a visual view of the runtime behavior of an *application*, not just interrupt signaling.

Dalton et al. [11, 13] present a toolkit for visualizing the local and distributed behavior of TinyOS Applications. Their tool uses the *nesC Analysis and Instrumentation Toolkit* (nAIT) [12] to instrument a target application. Probes inserted as part of the instrumentation process record runtime execution information. Trace data collected from an instrumented application is used to generate call graphs and sequence diagrams. Unlike the work presented by Dalton et al., the visual abstractions in our work are fundamentally different. In particular, our tool provides a higher-level abstraction that supports audience-specific configurability. (The visual representation in our work is configurable through an XML specification). Our tool provides an animated play-back mechanism for events occurring during execution and displays their effect on program state.

Finally, McCartney et al. [25] and Sallai et al. [35] describe TinyOS development environments that support syntax highlighting, code navigation, code completion, etc., to aid developers in understanding the static structure of TinyOS applications. These, however, do not provide insight into the runtime behavior of programs — our focus.

Chapter 3

TinyOS and nesC

Before proceeding to the design of the visualization framework, it is useful to briefly review the TinyOS programming model. In TinyOS, applications are written in *nesC* [16], a component-based variant of the C programming language. Applications are structured as a collection of *modules*, which bear similarity to object-oriented classes. In particular, modules encapsulate private state and behavior and communicate through well-defined interfaces. Both modules and interfaces may be parameterized to support generic types, similar to the linguistic features provided by Java and C++.

Each declared interface defines a set of *commands* and *events*. To understand the difference between the two types of operations, consider an interface `I` that declares a command `c()` and an event `e()`. Command `c()` may be invoked on a module that provides `I`, whereas event `e()` may be invoked on a module that uses `I`. Hence, commands impose implementation requirements on providers of the declaring interface, while events impose implementation requirements on users of the declaring interface. (In this sense, interfaces in nesC are similar to those in C#.) The idiomatic expression of a *split-phase operation* involves the declaration of a command used to initiate the operation and an event used to signal its completion.

One important point to emphasize is that module implementations in nesC are fully decoupled; one never references another by name. Instead, each module declares the interfaces it *provides* to other components, as well as the interfaces it *uses* to implement its behavior. To construct a complete application, each module that uses a given interface must be bound to a module that provides that interface. This binding is expressed in a nesC *configuration* comprising statements

```
 1 #include "Timer.h"
 2
 3 module BlinkC {
 4   uses interface Timer<TMilli> as Timer0;
 5   uses interface Timer<TMilli> as Timer1;
 6   uses interface Timer<TMilli> as Timer2;
 7   uses interface Leds;
 8   uses interface Boot;
 9 }
10 implementation {
11   event void Boot.booted() {
12     call Timer0.startPeriodic( 250 );
13     call Timer1.startPeriodic( 500 );
14     call Timer2.startPeriodic( 1000 );
15   }
16
17   event void Timer0.fired() {
18     call Leds.led0Toggle();
19   }
20
21   event void Timer1.fired() {
22     call Leds.led1Toggle();
23   }
24
25   event void Timer2.fired(){
26     call Leds.led2Toggle();
27   }
28 }
```

Listing 3.1: Blink Module

that match interface users to interface providers.

In the following section, we provide an example to make these concepts more concrete.

3.1 Example Application: Blink

We consider an example from the standard TinyOS distribution. The application starts three timers (that fire periodically), each at a different time. The LEDS located on the device toggle every time a timer is fired. Listing 3.1 shows the implementation of the BlinkC module. The component uses three interfaces — Boot, Leds, and three instances of the Timer interface (lines 4–8). Timer is parameterized by the type TMilli which represents the precision of the Timer as millisecond. nesC allows interface names to be locally renamed so that multiple instances can be used. As noted previously, the interfaces define the commands that can be used by BlinkC module and events that

```
1 configuration BlinkAppC {
2 }
3 implementation {
4   components MainC as MainC, BlinkC as MyBlinkC, LedsC as LedsC;
5   components new TimerMilliC() as Timer0;
6   components new TimerMilliC() as Timer1;
7   components new TimerMilliC() as Timer2;
8
9   MyBlinkC.Boot -> MainC.Boot;
10  MyBlinkC.Timer0 -> Timer0;
11  MyBlinkC.Timer1 -> Timer1;
12  MyBlinkC.Timer2 -> Timer2;
13  MyBlinkC.Leds -> LedsC;
14 }
```

Listing 3.2: Blink Configuration

must be implemented by the module.

Lines 11–15 show the event Boot.booted(). This event is defined as part of the Boot interface and is signaled when the device initializes. In the implementation of this event, the startPeriodic() command is called to initiate a periodic timer on the three Timer instances. The command accepts as arguments the frequency at which these timers are to be initiated.

After every quarter of a second, the TinyOS operating system signals a fired() event on the instance Timer0 and the red LED on the device is toggled. The handler is shown on lines 17–19. Timer1 and Timer2 are started every half and one second, and the events Timer1.fired() and Timer2.fired() are invoked, respectively. These two methods then toggle the green and blue LEDS, respectively. The two events are shown on lines 21–27

Listing 3.2 shows the top-level configuration file of the Blink application. Lines 4–7 list the modules that are required to bind the used interfaces with their respective providing components. MainC initializes all the system components and provides the Boot interface. The BlinkC module is locally renamed to MyBlinkC. The configuration also declares the LedsC component. This component implements the commands associated with the Leds interface. Generic components are instantiated using the keyword *new*. The component TimerMilliC implements the Timer interface and is locally renamed to Timer0, Timer1, and Timer2. Next, lines 9–13 show the *wirings* of the application. On line 9, the Boot interface used by MyBlinkC is wired to the MainC component. On lines 10–12, the three instances of Timer used by MyBlinkC module are wired to the three TimerMilliC instances. Finally, line 13 shows the wiring of the Leds interface used by MyBlinkC to the component LedsC.

The same application would look different in any imperative programming language. For example, in JAVA, it is not possible to rename interfaces (separate variables are used to reference different objects). Also, the standard programming languages follow blocking execution semantics. Linear execution of code can wait for an operation that takes long time to complete. As a result, there are no asynchronous event handlers. This makes program comprehension easier. However, in nesC, the programming model follows a non-blocking execution mechanism making it difficult for developers to understand program control flow.

Chapter 4

Framework Design

In this chapter, we introduce the animation system developed to support the visualization of TinyOS programs. As noted at the outset, the system uses an XML specification to identify program elements and the display properties of the visualization. It then presents an animated representation of runtime program behavior. The system can be used to understand both local and distributed behavior. We first present an overview of the framework design in section 4.1. We then proceed to describe the individual components of the framework in the subsequent sections.

4.1 Overview

An overview of the framework architecture is shown in Figure 4.1. The central component of the framework is the *Analyzer*(3), which accepts the top-level configuration of a nesC source application, along with an XML specification that identifies the program elements (e.g., function invocations, state relations) the user wishes to visualize. The *nesC Analysis and Instrumentation Toolkit*(nAIT) [12] parses nesC source materials, constructs an abstract syntax tree representation of the code and provides API that enables users to add program fragments to the tree. The Analyzer uses the API exposed by the nAIT(2) to insert probes within the identified operation set. These probes capture function invocations, returns, and variables that correspond to a change in the program state. Most of the probes record string data (e.g. function names, variable names). Recording explicit strings would, however, be inefficient as it would require increased storage space on a memory-constrained device. To address this problem, the probes record numeric constants

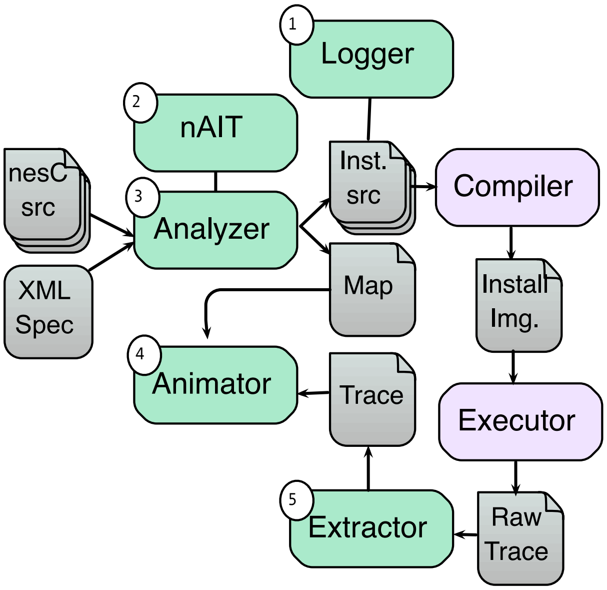


Figure 4.1: Framework Overview

mapped to string values. The Analyzer generates a map and uses it to associate strings corresponding to program elements with numeric constants. This map is stored on a base station and is later used in reassembling a recorded trace for the purposes of visualization. Each probe registers the numeric constants associated with the strings corresponding to program elements using a call to the *Logging Service*(1), a buffered logger built over `LogStorageC` (included as part of TinyOS). The Analyzer then regenerates the instrumented source base, which is then compiled and executed on the target platform. After the desired execution period, the *Extractor*(5) installs an application on the device that retrieves the captured trace data and relays it to a base station via a serial port. (The Logging Service and the Extractor were adapted for use from the system described in [11] and were modified to suit the needs of this tool). The Extractor then reassembles the logged trace data by mapping the numeric constants contained in the trace to their corresponding strings using the information stored in the map. The *Animator*(4) uses the reconstructed trace to show an animated view of the events that occurred during execution of the application. The visual abstractions used in the animation, as well as the components of the framework, are described in the sections that follow.

Before proceeding, it is useful to consider the characteristics of the target hardware platforms. The work presented here targets *mote*-class embedded devices. Each device comprises a suite of sensors, a general-purpose microcontroller, non-volatile storage, a wireless transceiver, and a power supply. Although the visualization framework is applicable to any TinyOS-supported device, the implementation presented here has been tested on the *Tmote Sky* platform [28]. This matchbox-sized device includes an 8Mhz MCU with 48K of ROM and 10K of RAM, 1Mb of EEPROM storage, and a 2.4Ghz Zigbee transceiver.

4.2 XML Specification

Each XML specification identifies the elements to be visualized and the display properties of the visualization. Listing 4.1 shows a typical specification[1]. The root of the specification, `rootComponent`, is the top-level module of the input application, rendered as a rectangle at the center of the animation screen (line 3). The attribute `name` specifies the module name, and `bgColor` specifies the color in which the top-level module will be shown during visualization. The element `networkSize`, shown on line 4, specifies the number of nodes that are part of the visualization. For visualizing local behavior, the `networkSize` element specifies the value one. The element `topModuleRendering` (lines 5–37) specifies how the state and behavioral aspects of the top-level module are to be displayed. The `State` element (lines 6–11) can contain several `iconImage` elements that allow the user to specify nesC code fragments. The fragments return an *image identifier* (line 7–9). Each image identifier corresponds to an image path on the base station; the image is intended to represent a particular runtime state of the module. Strings in the code fragments corresponding to image paths are replaced with numeric constants by the Analyzer and stored as a map. These processed fragments are later inserted into the application code as part of the instrumentation process. When trace data is reassembled at the base station, the Extractor retrieves the numeric constants from the recorded trace and performs a lookup in the stored map to find their corresponding string values. These strings are used to return the corresponding images. The Animator (later) displays the images at appropriate locations on the screen to indicate a state change. The element `watchWindow` (lines 12–18) is used to present additional information about program state. A watch window element can have one or more `stateComponents`. Each `stateComponent element` has an attribute `name` that specifies the name of the watch component and contains an arbitrary code block (lines 13–16). The code block returns a value based on conditions of the current program state, which is displayed during

---

[1] The XML syntax used in the current implementation of the tool differs somewhat from the format presented here. We present a simplified syntax for the sake of presentation.

visualization. In other words, the code block watches for changes occurring to program variables. As a simple example, for a program variable *busy*; the user can choose to display a value 1 when busy is true or 0 otherwise. This additional information is presented to the user during visualization in the form of a watch table.

On lines 19–36, the element `functions` specifies how the local functions of the top-level module are to be rendered during visualization. A `function`[2] element (20–34), displayed as a label in the box used to represent the top-level module, has attributes `name` and `labelColor` (line 20). These specify the function name and the background color of the function label, respectively. The `function` element contains the elements `invocation` and `return` which define the images shown during invocation and return of the function, respectively. These images are shown as moving from the label corresponding to the calling function, across the screen, to the target function and then back to the caller when the function returns. It is worth emphasizing that different images can be shown for a function invocation (or return) to reflect program state. In the listing, for example, the path to *image_1* is returned when *var_1* is true. A different image is returned otherwise (lines 21–33).

The element `usedInterfaceRendering` (lines 38–48) specifies the way interfaces used by the top-level module are displayed during visualization. The background color for these interfaces is specified by the `color` attribute. All these interfaces are shown as rectangular boxes rendered in a circle centered around the box that represents the top-level module. Again, all functions provided by these interfaces that should be visualized are specified by the `functions` element. Each function provided by an interface is displayed as a label inside the rectangular box associated with the interface. However, in this case, the `name` attribute of the `function` element denotes the fully qualified name–the providing interface and the function name (line 40). This helps in distinguishing functions invoked on different instances of an interface.

Finally, it is worth emphasizing that the XML specification inherently supports filtering to provide multi-resolution views of program state and behavior. A user can ignore information about program state by not specifying `iconImage` and/or `stateComponent` tags, etc., and thus limit their visualization focus. This enables users to focus on the visualization of specific program parts, that which are of most interest to the user.

---

[2] There is no distinction between events, commands, or tasks at this level; we refer to all of these elements as functions.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE visualization SYSTEM "AnimationSpec.dtd">
<rootComponent name="module_name" bgColor="white">
  <networkSize> 1 </networkSize>
  <topModuleRendering>
    <state>
      <iconImage>if(condition_1) return(image_path_1);
                 else return(image_path_2);
      </iconImage>
      ...
    </state>
    <watchWindow>
      <stateComponent name="Name_1">
        if(condition) return(value1);
        else return(value2);
      </stateComponent>
      ...
    </watchWindow>
    <functions>
       <function name="function_1" labelColor="white">
           <invocation>
             if(var_1 == true) return(image_path_3);
             else {
               if(call_a_function() == -1) {
                 return(image_path_4);
               } else {
                 return(image_path_5);
               } }
           </invocation>
           <return>
             if(var_2 == var_3) return(image_path_6);
             else return(image_path_7);
           </return>
       </function>
       ...
    </functions>
  </topModuleRendering>
  <usedInterfaceRendering color="white">
    <functions>
       <function name="iface.function_2" labelColor="white">
           <invocation> if(var_4 == 1) return(image_path_8);
                        else return(image_path_9);
           </invocation>
           <return>return(image_path_10);</return>
       </function>
       ...
    </functions>
  </usedInterfaceRendering>
</rootComponent>
```

Listing 4.1: XML Specification

4.3 nAIT

The nesC Analysis and Instrumentation Toolkit(nAIT) [12] is a hardware-independent, JAVA-based tool for parsing, analyzing, instrumenting, and generating nesC application source code. nAIT exposes a rich set of APIs that enable a user to (i) easily traverse the in-memory representation of a source base (as an abstract syntax tree) (ii) add program fragments, and (iii) modify program fragments. The API also exposes methods that regenerate an entire source base. An important modification method exposed by the tool is `instantiateGenerics()`, which traverses the source and eliminates generic configurations and components. The generic parameters are replaced with actual arguments; instantiated generic types are created. To uniquely identify newly instantiated components, a unique tag is added to the component name, and all component references are updated to reflect this change. nAIT also provides a graphical tool to visualize in-memory representations so that users can quickly understand application structure. The set of traversal methods aid in this process. The nAIT tool can also be used for locating elements of a similar type. For example, all interfaces provided by a component can be identified by a single method call. Generation methods provided by this tool enable users to add new program fragments into an existing source tree. Together, these methods simplify the instrumentation task. For a detailed discussion of nAIT, see [12].

4.4 Analyzer

The Analyzer is responsible for instrumenting user-selected program elements in an application with probes that record trace data. The input XML specification specifies the program elements the user is interested in displaying during visualization. These include function invocations, returns, and variables representing program state and additional information of variable values for the watch window. A path to this specification is passed as argument to the Analyzer, which provides an interface for the user to upload the top-level configuration of the application. The Analyzer then uses nAIT to perform a full application parse based on the uploaded configuration file, instantiates generic components, and returns an in-memory representation of the complete source base in the form of an abstract syntax tree. The following sections describe the instrumentation process for capturing program behavior and state respectively, of the program elements identified in the XML specification.

4.4.1 Behavior

The Analyzer begins by locating in the application source, the functions listed in the XML specification. The identified functions are instrumented with probes to capture invocations and returns. These probes register information about the name of the invoked function along with the image path corresponding to the invocation. Image paths corresponding to return of functions are also registered. Recall that these names are mapped to unique numeric constants for effective storage on flash and stored as part of a map on the base station. The probes use a call to the Logging Service to register these numeric constants.

Since it is hard to predict the actual execution sequence of events, providing information about the invocations and returns offers useful insight into runtime behavior. For this purpose, functions are inserted with instrumentation code to record their entry and exit points. A usual procedure follows instrumenting functions at the target site, i.e., when actual invocation happens. The work of Dalton et al. [11, 13] also follows an instrumentation strategy that instruments target sites alone. However, a target-based instrumentation strategy is not, by itself, sufficient for our purposes. In part, target-based instrumentation does not enable the instrumentation site to acquire information about the calling context. Consider, for example, the TimerMilliC component. TinyOS applications often use multiple instances of this generic component, suitably renamed — e.g., SenseTimer, TransmitTimer. In such an application, if the underlying TimerMilliC component is instrumented to capture incoming calls, all Timer.startPeriodic() calls recorded to the system trace will be indistinguishable. That is, it will be impossible to determine from a given trace whether a particular call was intended for SenseTimer or TransmitTimer. We supplement target site instrumentation with *call site* instrumentation. Instrumentation performed at the call site temporarily stores the calling instance name within a global variable. At the target site, the value stored in the global variable is retrieved and a probe recording function invocation saves this value. Note that call site instrumentation could by itself capture the relevant data. The strategy would be however be inefficient. In general, there are multiple call sites for a given target site. Adding code that records trace data at every call site would mean inserting more probes into the application than necessary. This would increase memory usage — a precious commodity on the target hardware platform.

Listing 4.2 shows the instrumentation procedure for instrumenting call sites, using the function Timer.startPeriodic() as an example. The value of the global variable MilliTimer, is set in the

```
1 command void RadioControl.startDone(error_t err) {
2   call VisState.registerInvocation(17, get_17_invocationImageId()); {
3       call VisState.setGlobalVar(16);
4       call MilliTimer.startPeriodic();
5       ...
6   }
7 }
```

Listing 4.2: Call Site Instrumentation

```
1 command void Timer.startPeriodic(uint32_t dt) {
2   call VisState.registerInvocation(call VisState.getGlobalVar(),
3                                    get_Timer_InvocationImageId()); {
4     ...
5     call VisState.registerReturn(call VisState.getGlobalVar(),
6                                  get_Timer_ReturnImageId());
7     return;
8   }
9 }
```

Listing 4.3: Target Site Instrumentation

body of `RadioControl.startDone()`. Line 2 shows the function invocation being recorded. `registerInvocation()` accepts two parameters; the first is a numeric constant that denotes the function name, and the second is a numeric constant that denotes the image that is to be displayed when the function is invoked. On line 3, the global variable is set before the timer is started on the instance `MilliTimer`[3]. Listing 4.3 shows the instrumentation at the target site. When `startPeriodic()` is invoked, the global value is retrieved and recorded along with the function name, as shown on lines 2 and 3. Finally, lines 5 and 6 show the return event of the function being recorded.

4.4.1.1 Capturing Message Exchanges

A user may choose to visualize packet transmission and reception events in a distributed network by specifying the number of nodes through the `networkSize` element of the XML specification. The user also chooses the functions responsible for the transmission and reception of a message. These include the functions `send()` and `sendDone()` defined by the `AMSend` interface and `receive()` defined by the `Receive` interface.

In TinyOS, the `message_t` buffer is used for all communication purposes. The buffer is imple-

---

[3] The nesC language does not support global variables. Instead, global variables are introduced as components using a variation on the Singleton pattern originally identified by the Gang of Four [15].

```
1 module CC2420ActiveMessageP {
2   ...
3   uses interface VisualizationState as VisualizationTrace;
4 }
5 implementation {
6   enum {
7     VIS_SIZE = 3,
8     CC2420_SIZE = MAC_HEADER_SIZE + MAC_FOOTER_SIZE + VIS_SIZE,
9   };
10   ...
11 }
```

Listing 4.4: `CC2420ActiveMessageP` module

mented as a nesC structure and its contents can be accessed via functions provided by components implementing the interfaces that abstract the communication services. For example, the `AMSend` interface provides commands to send messages (`send()`) and to cancel requested transmissions (`cancel()`). The interface also defines the (`sendDone()`) event which is signaled on successful completion of a send. The `Receive` interface provides an interface for message reception. It declares the `receive()` event. This event is signaled when a message is received. Both these interfaces also have commands to point to the messages' payload. The component `CC2420ActiveMessageP` is responsible for providing these interfaces on the *TMote Sky* platform.

To understand message exchanges between nodes in a network, we instrument send and receive messages. The framework makes use of a modified version of the TinyOS messaging libraries, adopted from [11]. The messaging structure is modified as follows: Before a send event, the payload of the message is populated with a constant that identifies the message. In the implementation of the `CC2420ActiveMessageP` component that provides the communication interfaces, a probe is inserted to record the address of the destination (contained in the message header) and the message constant. When a receive event occurs, the constant saved in the payload of the message is retrieved, and a probe records the address of the sender and the message constant. Listings 4.4 and 4.5 show the instrumentation of the `CC22420ActiveMessageP` module, specifically the procedure for capturing sends. First, the module uses the interface `VisualizationState` (line 3 of listing 4.4). This interface declares the commands used to record send and receive messages[4]. In lines 7 and 8, the size of the message structure is increased to accommodate the message constant. Listing 4.5 shows the

---

[4] The interface also declares commands to register function invocations, returns, capture state changes, watch window updates

```
 1 // Instrumenting send() in CC2420ActiveMessageP module
 2 command error_t AMSend.send[am_id_t id](am_addr_t addr, message_t* msg,
 3                                         uint8_t len) {
 4   ...
 5  call VisualizationTrace.registerSend(addr, len,
 6                                  call Packet.getPayload(msg,&len));
 7  return call SubSend.send( msg, len + CC2420_SIZE );
 8 }
 9
10 // Capturing send() in VisualizationStateC module
11 async command void VisualizationState.registerSend(
12                 am_addr_t dest, uint8_t len, void* payload) {
13  atomic {
14    uint16_t* localconstant;
15    if((len % 2) !=0) { // word-aligned boundary
16      len = len + 1;
17    }
18    localconstant = payload + len;
19    *localconstant = constantNumber++;
20     entry[bufIndex].entries[entryIndex].type = SEND;
21     entry[bufIndex].entries[entryIndex].te.radioEntry.addr = dest;
22     entry[bufIndex].entries[entryIndex].te.radioEntry.const =
23                                                      *localconstant;
24     updateOkay = addToLog();
25  } }
```

Listing 4.5: Recording Message Transmission

instrumentation of the send function. In lines 5–6, the registerSend() function of VisualizationState interface is called. It accepts the destination address, the length of the message and a pointer to the message payload as arguments. This is shown on lines 5–6. The implementation of registerSend is shown on lines 11–25. A constant identifying the message is added to the payload as shown on lines 18 and 19. This information is then recorded to flash storage (lines 20-24)

Capturing a receive message follows a similar procedure. When a message is received, the source of the message and the constant stored in the payload of the message are recorded to flash storage. Later, when trace data is extracted from flash storage, the address and the message constant are used to map send messages to their corresponding receives.

4.4.2 State

Recall that variables are monitored to capture state changes. More specifically, state elements that are material to the code fragments specified within the iconImage elements in the XML

```
1 // Code fragment from the XML specification
2 ...
3   <state>
4       <iconImage>
5         if(locked) return($(SRC)/misc/locked.jpg);
6         else return($(SRC)/misc/unlocked.jpg);
7       </iconImage>
8       ...
9   </state>
10 ...
11
12 // function added at module level
13 uint16_t getStateImageId() {
14     if (locked) {
15         return (0);
16     } else {
17         return (1);
18     } }
19
20 // function modifying the variable
21 event void  AMSend.sendDone(message_t *bufPtr, error_t error) {
22     call VisState.registerInvocation(2, get_2_InvocationId()); {
23         if (&packet == bufPtr) {
24             locked = FALSE;
25             call VisState.registerState(getStateImageId(), 0);
26             call VisState.registerWatch(getWatch_Name_1(), 0);
27         }
28         call VisState.registerReturn(2, get_2_ReturnId());
29         return;
30     } }
```

Listing 4.6: Recording Program State Change

configuration are monitored for changes. These variables affect the set of images displayed within the top-level module of the visualization. The Analyzer processes the code segment specified in each iconImage element to identify the variables used to compute the image paths returned by this tag. It then searches the source application for all functions which modify the identified variables. The Analyzer inserts the code fragments specified in the XML into the module as additional functions. After each variable modification (either due to an assignment operation or an increment/decrement operation), the newly added functions are called. The value returned by each of the functions is a numeric constant mapped to an image path. This constant is then recorded by the probes inserted by the Analyzer.

Listing 4.6 shows the code fragment specified in the iconImage element of the XML configuration file (lines 3–9) and the instrumented code (lines 12–30). This code fragment is used to

```
// Code fragment from the XML specification
...
  <watchWindow>
      <stateComponent name="Name_1">
        if(locked) return(25);else return(26);
      </stateComponent>
      ...
  </watchWindow>
...

// function added at module level
uint16_t getWatch_Name_1() {
    if (locked) {
        return (25);
    } else {
        return (26);
    } }

// function modifying the variable
event void  AMSend.sendDone(message_t *bufPtr, error_t error) {
    call VisState.registerInvocation(2, get_2_InvocationId()); {
        if (&packet == bufPtr) {
            locked = FALSE;
            call VisState.registerState(getStateImageId(), 0);
            call VisState.registerWatch(getWatch_Name_1(), 0);
        }
        call VisState.registerReturn(2, get_2_ReturnId());
        return;
    } }
```

Listing 4.7: Recording Watch Window Updates

capture a state change associated with a variable *locked*. The code fragment within the `iconImage` tag of the XML specification is processed and strings are replaced with numeric constants. The Analyzer then includes this processed code fragment within the function `getStateImageId()`. This function is inserted into the module and is shown on lines 13–18 of the listing. The function's return values, 0 and 1, correspond to the images *locked.jpg* and *unlocked.jpg*, respectively, and will eventually be displayed on screen. In the function `AMSend.sendDone()` (lines 21–30), the invocation of the function is recorded as shown on line 22. To record the variable image name (based on the value of the variable), a probe is added as shown on line 25. The `VisState.registerState()` function accepts the image identifier and the constant corresponding to the variable name as arguments. Line 28 shows the probe that records the return of `AMSend.sendDone()` event.

For the watch window display, a function containing the code fragment specified by each `stateComponent` element of the XML specification is inserted into the module. Listing 4.7 shows the

`WatchWindow` element of the XML specification and the instrumented source code. On lines 3–8, the `stateComponent` tag of the `WatchWindow` specifies that the user wants to display a value 25 when the variable *locked* is true and 26 otherwise. The user also specifies a name for the `stateComponent` as *Name_1*. This name is used while naming the function `getWatch_Name_1()` and helps in distinguishing each element of the watch window. The code fragment is included within the function body and inserted into the module as shown on lines 12–17. Lines 20–29 show the function that modifies the variable *locked*. In the body of this function, as with the case of capturing program state, a call is made to the newly added function (i.e., `getWatch_Name_1()`) and the return value is recorded by a probe. This is shown on line 25 of the listing.

After the instrumentation process, components that use the *Logging Service* are modified to add a wiring statement that binds the users of the service to its providing component `VisualizationStateC`. At this point, the Analyzer also saves information about the interfaces used by the top-level module. The information contains the names of the interfaces and the components providing these interfaces. When rendering the display, the boxes representing the used-interfaces are labeled using this information. The instrumented source is then regenerated, compiled, and installed on a device.

4.5 Trace Capture

During execution, events are recorded to the flash storage of the device. The Logging Service, adopted from [11], provides commands to log invocation, return, state events, watch window events, message sends and message receives. To efficiently record all events, the implementation follows a dual-buffering strategy. A buffer is written to flash storage when it fills up, while another buffer records the execution events on the device. Figure 4.2 shows a subsequence of an example trace recorded to flash when message exchanges are not recorded. The letters *i, r, s,* and *w* denote function invocation record, function return record, state change record, and watch window record, respectively. All entries are 4 bytes long. Each invocation, return, and state record stores the type of the record, the numeric constant associated with the name of the function or variable identified in the XML, and another numeric constant mapped to the display image representing the corresponding function invocation/return or a variable name. This image is shown as part of the animation. Since a watch window does not display an image, a watch window record stores the type of the record,

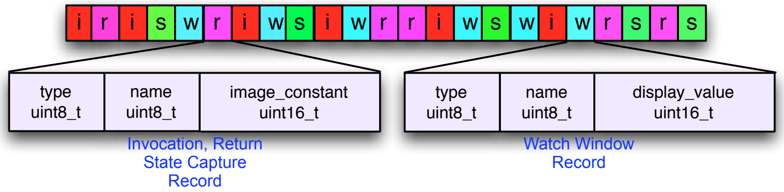


Figure 4.2: Trace Structure (Local Visualization)

name assigned to the watch component, and the associated display value corresponding to the return value of the arbitrary code fragment.

Figure 4.3 shows a subsequence of trace entries written to flash storge when message exchanges are logged. As noted previously, tracing message exchanges helps in understanding the behavior of message exchanges. The additional letters *t* and *a* represent send and receive message records, respectively. A send message entry logs the type of the record, the address of the destination (i.e., the node address to which the message is sent) and a message constant. A receive message entry logs the type of the record, the address from which the received message originated and the message constant. Send and receive records are also of 4 bytes.

After the application executes for a desired amount of time, the records written to flash are retrieved by the Extractor, also adapted from [11]. For a function invocation, return, state, and watch window records, the extracted data is reassembled to replace the recorded numeric constants with their corresponding string values. The reassembled trace data contains the names of functions that were invoked during execution, their corresponding invocation and return images, names of variables that caused a change in program state and the images used to represent the states. The reassembled trace also contains the names and corresponding display values associated with the watch window. For trace entries associated with message exchange, the source (and destination) address and message constant are used by the Animator described in section 4.6 to map send events to their corresponding receive events. This information is used to display a play-back of the system's execution.

Note that the trace storage structure could be generated dynamically to reduce the number of bits required to store the data of interest. For example, if only two functions are selected for visualization, then only two bits are required to record function names. We plan to integrate this capability in our future work. This will allow more records to be logged to flash storage, thus

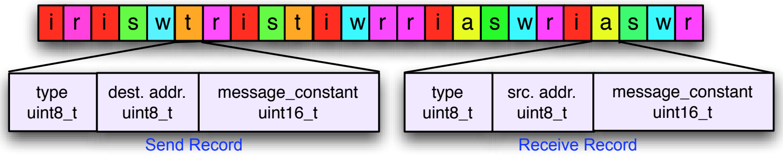


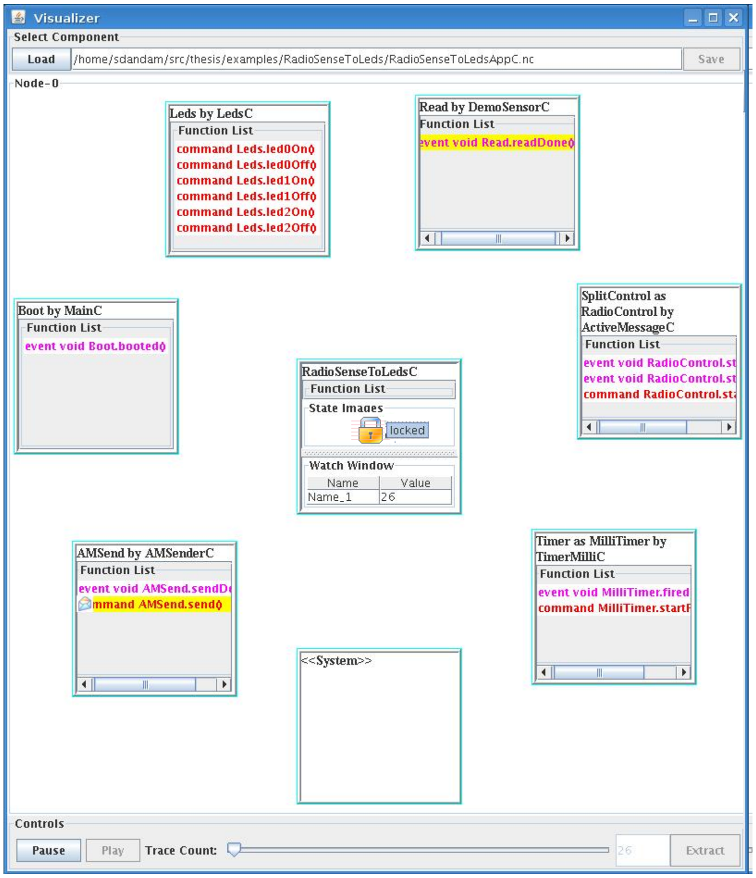
Figure 4.3: Trace Structure (Distributed Visualization)

enabling the user to capture more runtime events.

4.6 Animator

Figure 4.4 shows the rendered visualization for the application `RadioSenseToLeds`, included with the standard TinyOS distribution. A user clicks the load button to upload the top-level configuration file. The analyzer then parses the complete source base and adds instrumentation statements to capture the execution path. When the user clicks the *Save* button, the instrumented source base is regenerated, compiled and executed on the target platform. Also, at this point, the display is rendered on screen based on the program elements identified in the XML. As noted previously, the top-level module of the application is rendered as a rectangle at the center of the screen. Each interface used by the top-level module is shown as a separate rectangular box. A function provided by an interface is shown as a label inside the box representing the interface. Function names are color coded. Events are shown in magenta, commands in red, tasks in blue, and local module functions in black. The display is formatted so that all interface boxes form the outer circle and the box in the center represents the top-level module, the root of the XML specification.

Additionally, the `System` rectangle at the bottom of the display provides a way to divide the system behavior between what the user is interested in, and elements that are outside the scope of the users' interest. In effect, the display captures the state and behavior of interest, condensing everything else into a single component. Let us consider an example. A user chooses to visualize the functions `A()`, `B()`, and `C()` as part of her XML specification. The call sequence of the functions are: when the target device boots, it signals `A()`, which inturn calls `B()`. Function `B()`, inturn calls function `D()`. This function is not chosen by the user as part of her XML specification. Function `D()` makes a call to `C()`. The underlying function making a call to `A()` and the function `D()` constitute the `System`.



/home/sdandam/src/thesis/examples/RadioSenseToLeds/RadioSenseToLedsAppC.nc

Name	Value
Name_1	26

Trace Count: 26

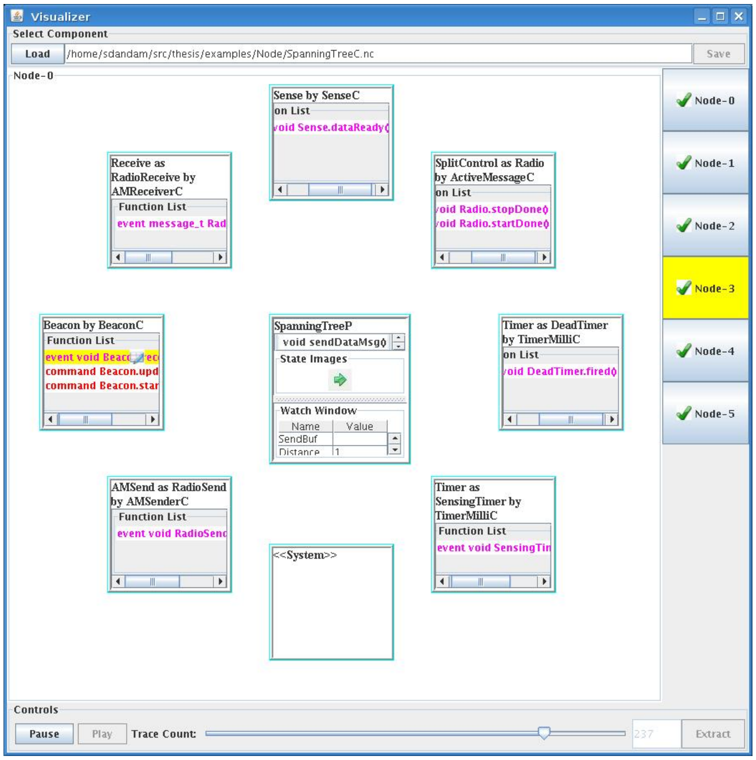
Figure 4.4: Local Visualization of RadioSenseToLedsAppC

The interface boxes are labeled with the interface name and the name of the providing component. The center box is labeled with the module name. Variables are shown as images rendered within the *State Images* section of the top-level module; changes in state are shown by a change in the display image(s), as specified in the XML. The watch window is a tabular representation of the name and display values corresponding to program state. Local functions are shown as part of the function list. The images shown in the figure represent either function invocations, returns, or state changes. In Figure 4.4, for example, an open envelope corresponds to the invocation image for the command `AMSend.send()`, while the image resembling a lock represents the change in program state when the value of the *locked* variable changes to true. The interface names, function names, and variable names shown on the screen are obtained from the XML specification. The slider at the bottom of the screen allows the user to visualize events that occurred before or after the current event by moving the slider to a desired location.

The display rendered so far is at the startup of the Animation System. The installed instrumented source base executes for a desired amount of time. When the user clicks the *Extract* button, trace data is extracted by the Extractor and reassembled. When the user clicks *Play*, for a function invocation, the animation renders the function invocation image as moving across the screen from the caller function to the callee function. A function return is shown by another image that returns back to the caller. A function is highlighted in *yellow* when it is called; all *active calls* remain highlighted (i.e., on the call stack). This provides insight into the call sequence of functions, which is otherwise difficult to understand.

4.6.1 Distributed Behavior

To present a visual representation of the distributed behavior of an application, the display is modified. Figure 4.5 shows the rendered visualization for a distributed case. First, the right-most column in the display shows the nodes in the network, along with their addresses. Each node is represented as a *button*. In this case, the node visualizations resemble a deck of cards. By default, the node with address zero is shown on the top. All other displays are hidden beneath the top-most node display, and a user can choose to display the visualizations by clicking the button corresponding to the node. The title at the left-hand corner displays the node name. The rendered visualization display for each node is identical to that described in the previous section and presents a local view for the selected node.



/home/sdandam/src/thesis/examples/Node/SpanningTreeC.nc

Name	Value
SendBuf	
Distance	1

Figure 4.5: Distributed Visualization of `SpanningTreeP`

To render trace entries associated with message exchange, the Animator maps send events to their corresponding receive events. When a message associated with send entry is encountered, it is added to *messages in transit* set. The send event is then shown in the visualization with an image that moves from the caller to the function `send()`. The receiving node is highlighted in yellow. On encountering a receive entry, the messages in transit set is searched to find a corresponding send and is matched based on the address of the sender and the messages' identifier. If the receiving node is at the top of the display, the receive event is shown on the screen as an image moving from the source node (shown on the right side of the display) to the receive function of the current node. In the figure, Node-0 has received an beacon message from Node-1. An image resembling an open book is shown moving the the button representing Node-1 to the function `Beacon.received()`. The image corresponds to the invocation image of the `received()` function as specified by the user in her XML specification. Otherwise, the receiving node is highlighted in *yellow*. In the figure, Node-3 has received a message from one of its neighboring nodes and is therefore highlighted. The user can choose to click on the highlighted node to continue visualizing application behavior at the receiving end. The highlighting of the node is removed as soon as the receive event is rendered on that particular node display (In other words, the node remains highlighted until a matching send is encountered). Also, the Animator maintains a list of received messages for every send message and updates this list after rendering the receive event. The send message is then removed from the messages in transit set after all of the receives corresponding to the send are rendered on screen (In case of a broadcast message, there are multiple receives for a single send event). The process continues until all recorded events are rendered on screen. The rendered visualization helps in understanding message exchanges between nodes in a network.

The animation on screen also shows the visualization of local behavior of a node. This includes the invocation and returns for selected functions, changes in program state and updates to the watch window. The animations for the nodes with hidden display happen instantly. This allows the user to see a continuous view of animations rather than wait for the animations to complete on hidden nodes. In Section 5.2, we discuss an application to understand distributed behavior.

A user may choose to focus on a small set of state components and function behaviors to support program understanding related to a specific reasoning task. The user might then choose to focus on a more a detailed set of state components and behaviors to provide finer-grained understanding. In fact, this suggests a common usage model: First, a user may use the tool to support a high-level understanding. In successive runs of the application, the XML specification might be refined to provide progressively more detail until the desired level of understanding is achieved.

Chapter 5

Use-case Scenarios

In this chapter we present two use-cases to demonstrate the utility of the animation framework. The use-cases illustrate the features of the framework and outline how the framework helps in understanding program behavior. First we use the `RadioSenseToLeds` application bundled with the standard TinyOS to demonstrate local visualization. The second example is a *self-stabilizing spanning tree* application[1] to exemplify distributed behavior of TinyOS programs.

5.1 Local Visualization

We first consider the `RadioSenseToLeds` application to demonstrate the framework's utility in understanding local program behavior. The application samples a sensor value periodically and broadcasts this value to neighboring nodes. The radio is started when the device boots. Receiving nodes display the last three bits of the received packet by toggling the LEDS on the device. Listings 5.1 and 5.2 show the XML specification of the application. The specification identifies the program elements the user is interested in displaying during visualization. These include function invocations, returns, and variables representing program state. Line 3 of listing 5.1 specifies the name of the top-level module and the background color in which the module is rendered on screen. Since the user is interested in visualizing local behavior, the `networkSize` is specified as one on line 4. Lines 5–19 of the listing specify the display properties of the elements that will be rendered in the top-level module box. The user wants different images (locked.jpg and unlocked.jpg) to be

---

[1] This application was developed as part of the course CpSc 881 during Fall 2007 by Sally K. Wahba and Sravanthi Dandamudi.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE visualization SYSTEM "AnimationSpec.dtd">
3 <rootComponent name="RadioSenseToLedsC" bgColor="gray">
4 <networkSize>1</networkSize>
5   <topModuleRendering>
6     <state>
7       <iconImage>
8         if(locked) return($(SRC)/misc/locked.jpg);
9         else return($(SRC)/misc/unlocked.jpg);
10      </iconImage>
11      ...
12    </state>
13    <watchWindow>
14      <stateComponent name="Name_1">
15        if(locked) return(25);else return(26);
16      </stateComponent>
17      ...
18    </watchWindow>
19  </topModuleRendering>
20  ...
21 </rootComponent>
```

Listing 5.1: XML Specification for RadioSenseToLeds

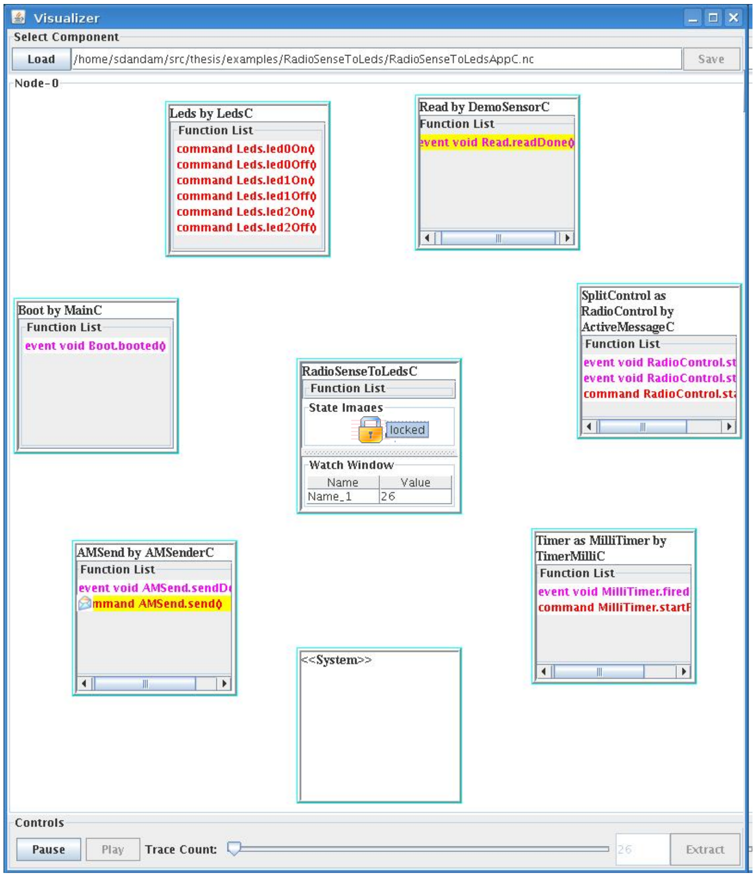
displayed based on the value of the variable *locked*. This is specified using the iconImage tag (lines 7–10). Users can also specify multiple iconImage tags. Each tag returns an image path representing a particular program state. Next, the watchWindow element is used to specify additional information that will be displayed to the user through the watch window section of the top-level module (lines 13–18). In lines 14–16, the user specifies a name for the watch window and values to be displayed. After the program elements and display properties of the top-module are specified, the user specifies the functions she wishes to visualize. These functions are provided by interfaces used by the top-level module. Listing 5.2 shows the remainder of the specification. In line 5, the user specifies that she wishes to display all used interfaces in gray. The functions element (lines 6–28) contains a list of functions[2] that would be included in the visualization. For example, the user identifies the function AMSend.send() and desires to display an image resembling an open envelope to show invocation and an image resembling a lock for the corresponding return (lines 7–10). Similarly, for the Read.readDone(), the user specifies a path to an open book image for invocation and a path to a closed book image to show the return (lines 11–14). Lines 15–26 show the specification for the functions AMSend.sendDone(), RadioControl.startDone(), and MilliTimer.startPeriodic().

---

[2] A partial set of selected functions is shown for the sake of presentation.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE visualization SYSTEM "AnimationSpec.dtd">
<rootComponent name="RadioSenseToLedsC" bgColor="gray">
  ...
  <usedInterfaceRendering color="gray">
    <functions>
      <function name="AMSend.send" labelColor="white">
       <invocation>return($(SRC)/images/open_envelope.jpg);</invocation>
       <return>if(locked) return ($(SRC)/misc/locked.jpg);</return>
      </function>
      <function name="Read.readDone" labelColor="white">
       <invocation>return($(SRC)/images/open_book.jpg);</invocation>
       <return>return ($(SRC)/images/closed_book.jpg);</return>
      </function>
      <function name="AMSend.sendDone" labelColor="white">
       <invocation>return($(SRC)/images/timerImg.jpg);</invocation>
       <return>return ($(SRC)/images/closed_envelope.jpg);</return>
      </function>
      <function name="RadioControl.startDone" labelColor="white">
       <invocation>return($(SRC)/images/signal_transmitter.jpg);</invocation>
       <return>return ($(SRC)/images/active_transmitter.jpg);</return>
      </function>
      <function name="MilliTimer.startPeriodic" labelColor="white">
       <invocation>return($(SRC)/images/clock.jpg);</invocation>
       <return>return ($(SRC)/images/default.jpg);</return>
      </function>
       ...
    </functions>
  </usedInterfaceRendering>
</rootComponent>
```

Listing 5.2: XML Specification for RadioSenseToLeds (Cont'd)



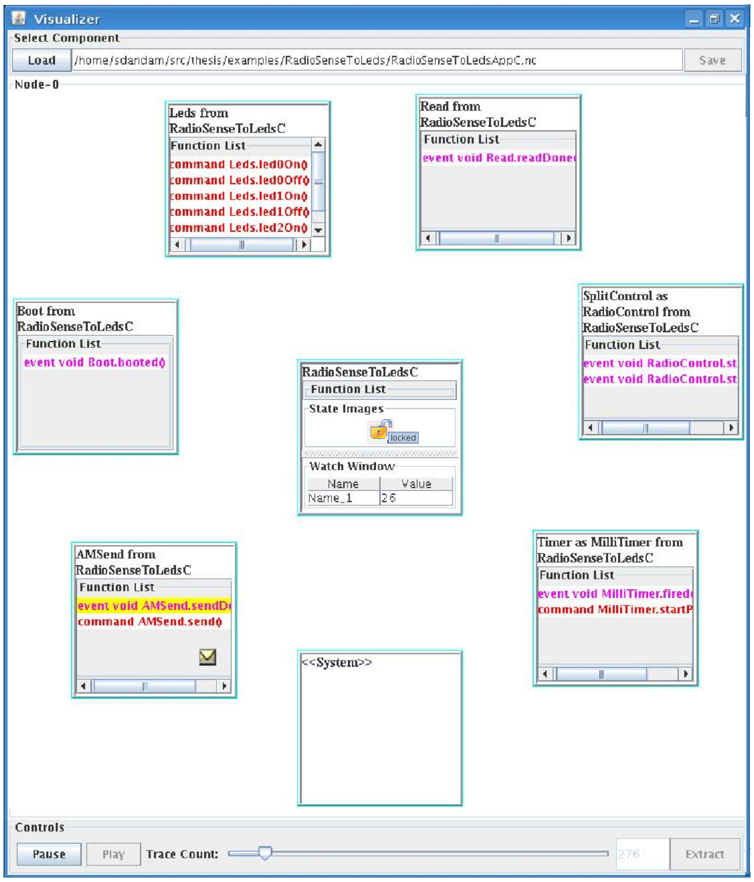
/home/sdandam/src/thesis/examples/RadioSenseToLeds/RadioSenseToLedsAppC.nc

Name	Value
Name_1	26

Trace Count: 26

Figure 5.1: `RadioSenseToLedsAppC` (showing a successful `send()`)

At start-up of the tool, a path to the XML specification is passed as an argument to the Analyzer. The Analyzer provides an user-interface to upload the top-level configuration file. The user then clicks the *Load* button, selects the configuration file and uploads it — `RadioSenseToLedsAppC.nc` in our example. The Analyzer now performs a complete parse of the source and inserts instrumentation statements as described in section 4.4. When the user clicks the *Save* button, the static portion of the visualization is rendered on screen. Figures 5.1 and 5.2 show the rendered visualizations of `RadioSenseToLeds`. As described in section 4.6, the top-level module of the application is rendered as a rectangular box in the center of the screen. The interfaces used by this module form the outer circle. The function list section of the top-level module box is empty in the figures as there are no local functions in `RadioSenseToLedsC`. Meanwhile, the instrumented application is compiled and executed on the target platform. The user allows the application to run for a desired amount of time and then clicks the *Extract* button. Trace data recorded to EEPROM is extracted and



/home/sdandam/src/thesis/examples/RadioSenseToLeds/RadioSenseToLedsAppC.nc

Name	Value
Name_1	26

Figure 5.2: RadioSenseToLedsAppC (showing return of sendDone())

reassembled at the base station. When the user clicks *Play*, an animation of the system execution is displayed to the user.

In figure 5.1, the Read.readDone() event has been signaled by the System, which then calls AMSend.send(). The invocation image of Read.readDone() is shown moving across the screen from the System until it reaches the function. The Animator highlights the function to indicate that it has been called. Then, the invocation image corresponding to AMSend.send(), resembling an open envelope, moves across the screen from Read.readDone() until it reaches AMSend.send(), which is then highlighted. The application makes use of a variable *locked* as a flag to indicate that a send is in progress. This variable is set to *true* if the function AMSend.send() returns success. This is shown by the image depicting a lock in the state panel of the top-level module. In figure 5.2, after a message broadcast, the System signals the AMSend.sendDone() event. The function is highlighted to show its invocation. Also, the variable *locked* is set to *false* to indicate that the radio channel is free to be used. This is shown by displaying an *unlocked* image in the state panel. The figure shows the animated return image (shown as a closed envelope) while the function call returns to the `System`. The animation continues until all recorded events in the application are rendered on the screen one at a time.

This example demonstrates the framework's potential to help in understanding local program behavior. Users can focus on the visualization of specific program elements by specifying them in the XML specification or can use the tool for a high-level understanding.

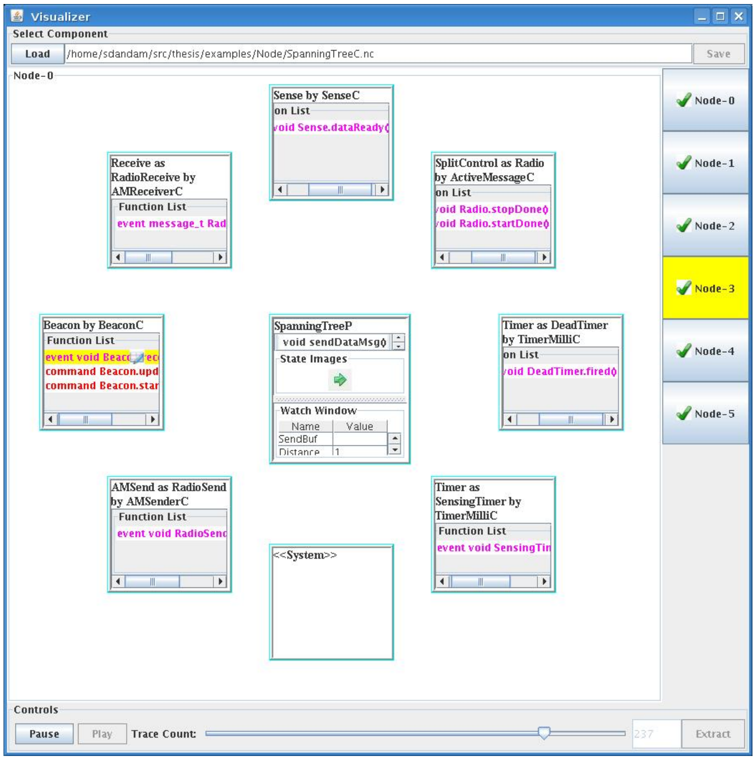
5.2 Distributed Visualization

We now turn our attention to illustrate the ability of the framework to aid in comprehending distributed behavior. We use a self-stabilizing spanning tree application for this purpose. Each node in the network acts as a sensing agent to sense four environmental parameters: humidity, temperature, total solar radiation, and photosynthetically active radiation. These values are routed to a basestation through a spanning tree. To achieve this goal, each node broadcasts a beacon message every five seconds. It then identifies a parent node based on the beacon messages received from other nodes. Sensed data is routed to the basestation via the parent. The XML specification for the application is similar to the specification described in the previous section. The user chooses to visualize distributed behavior on six nodes. She specifies the size in the `networkSize` element of the XML specification. The user follows the same procedure to upload the top-level configuration file (by clicking the *load* button). Again, the Analyzer parses the application and instruments the source base. The instrumented source base in installed on the devices[3] when the user clicks *save*. After the application runs for a desired amount to time, recorded trace data is retrieved from the all devices and reassembled. Figures 5.3 and 5.4 show the displayed visualization for this application at different points in time.

In the figures, the right-most portion of the display shows the set of nodes on which the application was installed. Each button corresponds to a node; the label denotes the node's address. The user can click one of these buttons to visualize the local behavior of that particular node. When

---

[3] 80 *TMote sky* nodes are connected directly to the base-station that runs the Animation Framework and are arranged in a grid containing 8 rows of 10 nodes each. Based on the network size, the program is installed sequentially these nodes.



/home/sdandam/src/thesis/examples/Node/SpanningTreeC.nc

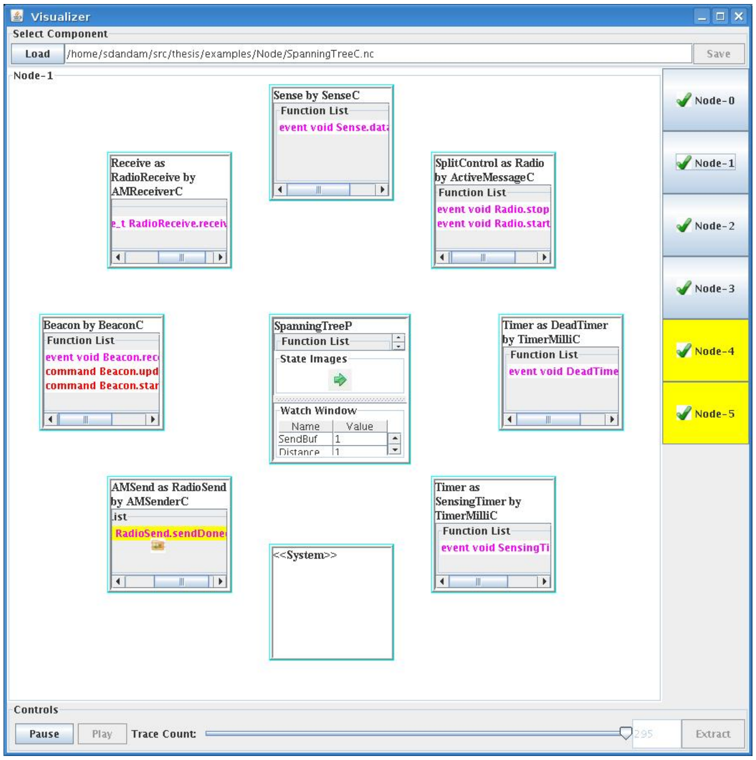
Name	Value
SendBuf	
Distance	1

Trace Count: 237

Figure 5.3: SpanningTreeP (showing receipt of a beacon from Node-1)

a message is sent from a node, the button corresponding to the receiving node is highlighted in yellow. The module contains `sendDataMsg()` declared as a local function and is listed in the *Function List* section of the top-level module. For example, in figure 5.3, Node-0 has received a beacon message from Node-1. This is shown by an image (resembling an open blue book) moving from Node-1 to the function `Beacon.received()`. The return of this function is shown by an image resembling a closed red book, moving from the function `Beacon.received()` to `System`. Meanwhile, Node-3 has received a message from one of its neighboring nodes. The node is highlighted until it is rendered on the corresponding node. Since Node-0 is on the "top" of the display deck, the animations representing the execution sequences of other nodes occur in the background. The user can switch to a different node at any time to continue visualizing program behavior of that node.

Figure 5.4 shows the visualization display with Node-1 on the top of the visualization deck. In the figure, the sensor readings of Node-1 have been sent to a parent node and the `RadioSend.sendDone()` event is fired on successful completion of send. Again, the parent node (Node-4, in this case), is highlighted to show the receipt of the message. We emphasize that as in the local



/home/sdandam/src/thesis/examples/Node/SpanningTreeC.nc

Name	Value
SendBuf	1
Distance	1

Figure 5.4: `SpanningTreeP` (showing `sendDone()` of sensor readings)

case, the images in the state panel of the top-level module change to reflect changes in program state. In this example, the variable *sendBufInUse* is used as a flag to indicate a send in progress. This variable is set to the value *false* when `RadioSend.sendDone()` is a success. The watch window shows the value 1 when the distance of the current node from the root is greater than or equal to 1. The value 0 implies that the node is itself the root. The framework provides a view of the distributed behavior of the application. The animations present system execution paths and provide insight about program control flow — which is otherwise hard to understand through manual tracing of source code.

Chapter 6

Performance Analysis

We now analyze the performance of the Animation Framework, focusing on the memory overhead introduced during the instrumentation process, the number of program events that may be logged to flash storage, and the capture rate of program events. Applications from the standard distribution of TinyOS are used as test cases. We follow an analytical process based on the approach followed in the work of Dalton et al. [13]. Though the analysis mirrors the work of Dalton et al. , the results naturally vary since the instrumentation strategy is different. Instrumenting both target and call sites introduces additional memory overhead. Also, this tool introduces probes that register variable names, image names that represent varying program states, and additional information for display purposes (later shown in the form of a watch window). These increase the number of events recorded to flash storage.

Table 6.1, Figure 6.1, and Figure 6.2 summarize the memory usage under four scenarios.

Application	*Baseline*		*EEPROM Logging*		*Minimum Instrumentation*		*Complete Instrumentation*	
	RAM	*ROM*	*RAM*	*ROM*	*RAM*	*ROM*	*RAM*	*ROM*
Null	4	1418	188	8374	592	8668	824	12470
Powerup	4	1472	188	8388	592	8682	824	12568
Blink	55	2654	218	8608	622	8914	852	13200
Sense	95	7184	258	12910	660	13220	886	17672
RadioCountToLeds	286	10794	428	15828	828	16140	1056	21874
RadioSenseToLeds	344	15158	480	20122	882	20488	1124	25746
Oscilloscope	372	15320	508	20264	914	20612	1156	26008
MViz	1722	33640	1860	38520	2260	38864	2287	42498
MultihopOscilloscope	3368	28624	3504	33446	3906	33910	3933	38250
MultihopOscilloscopeLqi	2451	24438	2593	29372	3906	33910	3933	37598

Table 6.1: Resource Overhead(RAM/ROM)

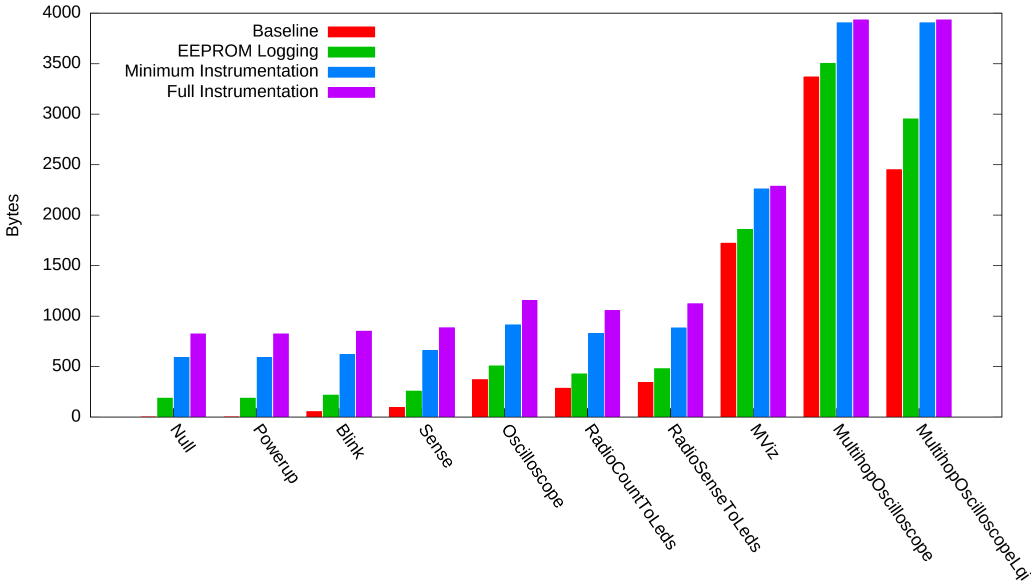


Figure 6.1: RAM Overhead

In Table 6.1, we first present the usage for the base application without any source modifications, shown in the first column. To understand the overhead introduced by our logging service, which includes `LogStorageC`, we present the RAM and ROM usage for `LogStrorageC` in the second column. For each application, a single entry is logged to flash to ensure that the compiler does not remove the logging component as part of its dead code elimination phase. Since the implementations of the applications remain the same, the results presented in both these columns are close to those presented by Dalton et al. [13]. We use the same set of applications and the standard logging service as used by Dalton et al. . However, since the underlying structure used to record events is different, there is a slight variation in memory usage for EEPROM logging. The third column shows the memory usage figures when the application uses the Logging Service of the Animation Framework to instrument one function — the function `Boot.booted()` is instrumented and the invocation and return events are recorded for all applications. The instrumented source in this case does not include any instantiated generic components, and therefore does not include the overhead caused by generic component instantiation. The final column shows the memory usage for a complete instrumentation of the application using the Logging Service. A complete instrumentation involves instantiating generics, inserting probes to record the invocations and returns of all functions provided by the interfaces used by the top-level module at both the call and target sites, send and receive messages, variable names, image names representing program state and additional display values for the watch

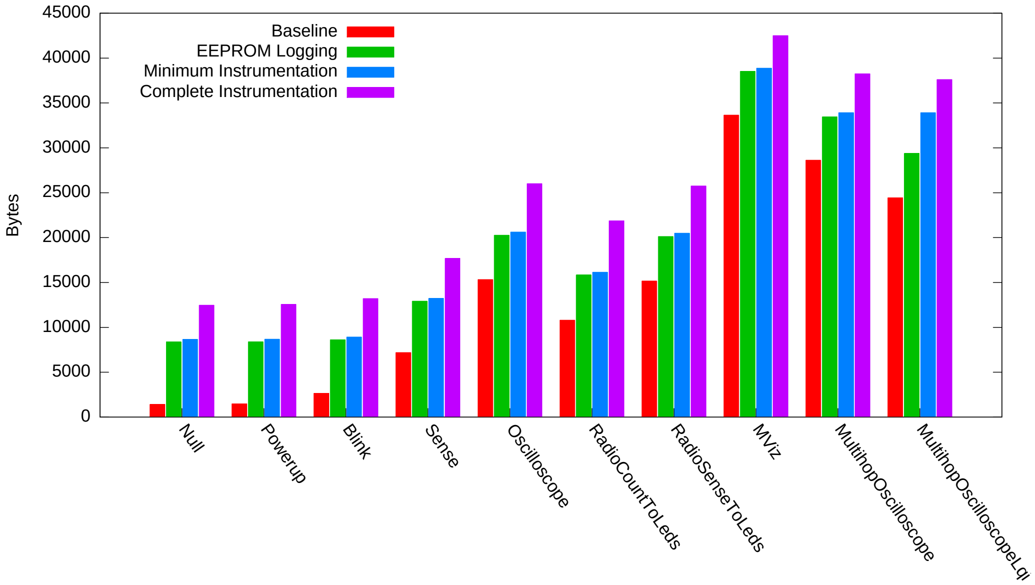


Figure 6.2: ROM Overhead

window. The complete instrumentation includes the modified messaging libraries which capture send and receive messages.

The results show that the average RAM and ROM overhead for the minimal instrumentation over the baseline is 546 and 6212 bytes, respectively. Complete instrumentation requires an average of 171 and 4445 additional bytes of RAM and ROM, respectively, over the minimal instrumentation. It can be seen from the figures and the table that for large applications like `MultihopOscilloscope`, only 27 additional bytes of RAM and 4340 bytes of ROM (over the minimal) are required. The figures also indicate that the total additional overhead on RAM and ROM for large applications like `MViz` and `MultihopOscilliscope` is small.

Figure 6.3 shows the incremental overhead on ROM and RAM for every call to the Logging Service. The `Blink` application is instrumented multiple times with an increasing number of calls to the Logging Service. A complete instrumentation of this application requires a total of eighteen calls to the Logging Service and include both call site and target site instrumentation. In the first case, only the event corresponding to the invocation of `Boot.booted()` is logged; in the second, invocation and return events corresponding to `Boot.booted` are logged. The invocation of a second function (`Timer1.fired()`) is also logged in the third case and so on. The incremental overhead also includes the overhead introduced by call site instrumentation. On average, each additional probe requires 29 bytes of ROM. There is no associated RAM expense. Again, for each of these logged events, the

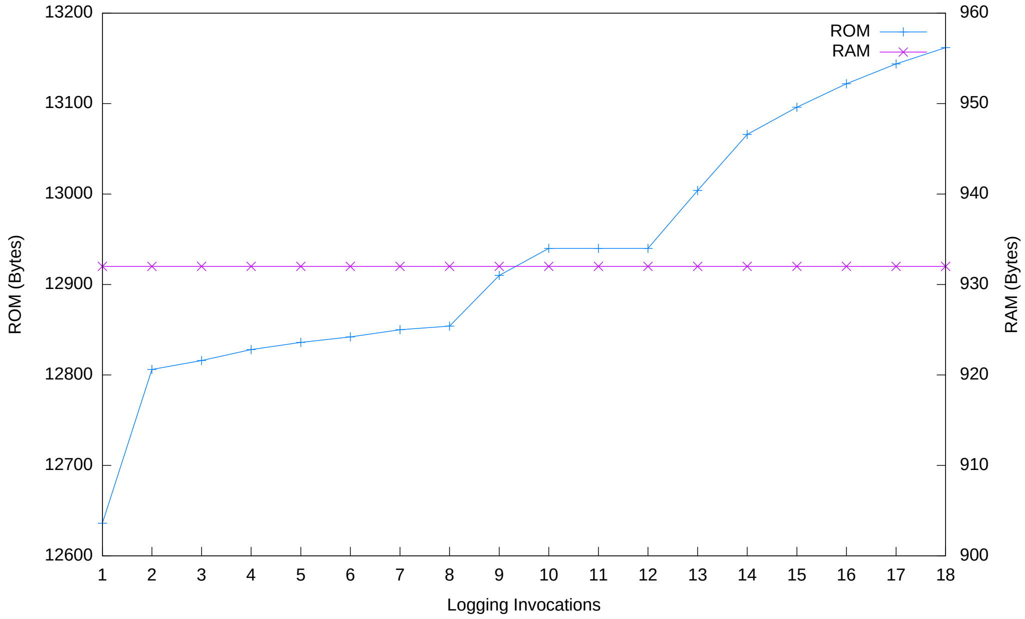


Figure 6.3: Incremental Overhead

Bytes	*No. of Buffers*	*Time (seconds)*			
		Minimum	*Maximum*	*Average*	*Standard Deviation*
4	1024	0.016	1.242	0.021	0.074

Table 6.2: Capture Rate

record size is 4 bytes. The average does not vary significantly for other applications in the test suite.

Though our tool inserts probes that record variable changes corresponding to state and watch window changes, the associated cost to insert one such probe is not different from that required to capture function invocations or returns. This is because the record structure used in both the cases is the same. In other words, in the minimal instrumentation scenario, instead of capturing the invocation of `Boot.booted()`, if the instrumentation process is modified to capture one change of a variable, the memory usage would remain as shown in the third column of table 6.1.

Increased instrumentation density will increase a user's ability to capture events occurring with high frequency, and will therefore enable fine-grained visualizations. Each record logged to flash storage requires 4 bytes; 60 entries are written at a time. Table 6.2 shows the amount of time required to record each buffer to flash storage. The second column in the table represents the total number of buffers that are written to flash before the log storage is full. It can be seen that the minimum time taken to record a completely filled buffer is very close to the average with a standard deviation of only 0.074 seconds. The maximum time it took to record 60 entries (of 4 bytes) to flash storage is 1.242 seconds. This means in one second, approximately 48 events can be triggered and recorded by the Logging Service. If all the space is used for logging, approximately 4368 complete buffers can be filled and written, which totals 262,080 events. These events, in most cases, are sufficient to provide a good understanding of an application.

Chapter 7

Conclusion

Wireless sensor networks enable the development of a rich class of applications ranging from those used to monitor the environment to traffic monitoring. These applications run on memory-constrained devices that consist of tiny sensors that sense and communicate environmental parameters to other devices. Though the programming language used for implementing these applications offers several advantages to cope with the event-driven nature of the applications, the programming model adopts asynchronous, split-phase execution semantics. Developers must manage program state distributed across various event handlers. These handlers execute operations and are themselves distributed across compilation units.

Our work is based on the observation that program understandability becomes difficult when developers must handle the inherent distributed nature of wireless applications and must design systems which adhere to a programming model that introduces non-determinism. They are tasked with explicit management of the control state that governs the behavior of event handlers — thus hindering their ability to understand execution behavior.

To assist developers in comprehending runtime behavior of wireless sensor applications, we described a framework for visualizing TinyOS programs. An XML-based configuration file is used to identify program elements and also the display properties of the visualization. The application is parsed and based on the XML specification, probes are inserted to record function calls and changes in program state to the EEPROM. Recorded trace data is extracted from the device to present an animated play-back like visualization that helps developers to comprehend local execution sequence of the program. The approach is extended to provide visualization support for understanding distributed behavior. In addition to recording function call information and changes in program state, the messaging libraries of TinyOS are modified to include a unique constant in every message. This constant is used to associate a send message with its corresponding receive messages. The send and receive messages are also recorded to EEPROM. When recorded trace data is extracted, the information helps correlating control-flow on different nodes. The visualization links causally related events on different nodes based on send and receive messages. We then presented an analysis of the performance of the Animation Framework. The implementation was evaluated based on additional memory required on the resource-constrained devices and the number of records logged to the flash storage.

We have described an Animation Framework for visualizing programs written for the TinyOS platform. We believe that this system will help users understand the runtime program behavior of wireless sensor networks. Developers can focus on the core functionality of their application and use this framework to understand program control-flow. The development of such visualization tools helps building and debugging of wireless sensor applications easier.

Bibliography

[1] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y.R. Choi, T. Herman, S.S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *Computer Networks*, 46(5):605–634, 2004.

[2] T.J. Ball and S.G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, 1996.

[3] E.A. Basha, S. Ravela, and D. Rus. Model-based monitoring for early warning flood detection. In *The 6th ACM Conference on Embedded Network Sensor Systems*, pages 295–308, New York, NY USA, November 2008. ACM.

[4] J. Beutel, S. Gruber, A. Hasler, R. Lim, A. Meier, C. Plessl, I. Talzi, L. Thiele, C. Tschudin, M. Woehrle, and M. Yeucel. PermaDAQ: A scientific instrument for precision sensing and data recovery in environmental extremes. In *The 8th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 265–276, New York, NY USA, April 2009. ACM.

[5] J. Bohnet and J. Döllner. Visual exploration of function call graphs for feature location in complex software systems. In *The Proceedings of the 2006 ACM symposium on Software visualization*, pages 95–104, New York, NY, USA, September 2006. ACM.

[6] K. Brade, M. Guzdial, M. Steckel, and E. Soloway. Whorf: A visualization tool for software maintenance. In *The Proceedings of the 1992 Workshop on Visual Languages*, pages 148–154, New York, NY, USA, September 1992. ACM.

[7] L.C. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Transactions on Software Engineering*, 32(9):642–663, 2006.

[8] M. H. Brown and R. Sedgewick. A system for algorithm animation. In *The 11th Conference on Computer Graphics and Interactive Techniques*, pages 177–186, New York, NY, USA, 1984. ACM.

[9] K. Chintalapudi, T. Fu, J. Paek, N. Kothari, S. Rangwala, J. Caffrey, R. Govindan, E. Johnson, and S. Masri. Monitoring civil structures with a wireless sensor network. *IEEE Internet Computing*, 10(2):26–34, 2006.

[10] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *The 15th IEEE International Conference on Program Comprehension*, pages 49–58, Washington, DC, USA, June 2007. IEEE.

[11] A.R. Dalton and J.O. Hallstrom. A toolkit for visualizing the runtime behavior of TinyOS applications. In *The 16th IEEE International Conference on Program Comprehension*, pages 43–52, Washington DC, USA, June 2008. IEEE.

[12] A.R. Dalton and J.O. Hallstrom. nait: A source analysis and instrumentation framework for nesc. *The Journal of Systems and Software (JSS)*, 82(7):1057–1212, 2009.

[13] A.R. Dalton, S.K. Wahba, S. Dandamudi, and J.O. Hallstrom. Visualizing the runtime behavior of embedded network systems: A toolkit for TinyOS. *Science of Computer Programming*, 74(7):446–469, 2009.

[14] G. G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, and M. Welsh. Monitoring volcanic eruptions with a wireless sensor network. In *The 2ndProceedings of the European Workshop on Wireless Sensor Networks*, pages 108–120, Los Alamitos, CA, USA, January–February 2005. IEEE Computer Society.

[15] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA USA, 1995.

[16] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D.E. Culler. The nesC language: A holistic approach to networked embedded systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–11, New York, NY USA, June 2003. ACM Press.

[17] P. V. Gestwicki. Interactive visualization of object-oriented programs. In *The 19th annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 48–49, New York, NY, USA, October 2004. ACM.

[18] G. Gracioli and S. Fischmeister. Tracing interrupts in embedded software. In *The 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 137–146, New York, NY, USA, June 2009. ACM.

[19] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D.E. Culler, and K. Pister. System architecture directions for networked sensors. In *The 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, New York, NY USA, November 2000. ACM Press.

[20] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, Sept.-Oct. 2006.

[21] X. Jiang, S. Dawson-Haggerty, P. Dutta, and D. Culler. Design and implementation of a high-fidelity AC metering network. In *The 8th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 253–264, New York, NY USA, April 2009. ACM.

[22] D.B. Lange and Y. Nakamura. Program Explorer: A program visualizer for C++. In *The USENIX Conference on Object-Oriented Technologies*, page cdrom. USENIX Association, June 1995.

[23] J. Li. Monitoring of component-based systems. Number HPL-2002-25(R.1), Palo Alto, CA, USA, May 2002. Imaging Systems Laboratory, HP Laboratories.

[24] B.A. Malloy and J.F. Power. Exploiting UML dynamic object modeling for the visualization of C++ programs. In *The 2005 ACM Symposium on Software Visualization*, pages 105–114, New York, NY, USA, May 2005. ACM.

[25] W.P. McCartney and N. Sridhar. TOSDev: A rapid development environment for TinyOS (demo). In *The 4^{th} International Conference on Embedded Networked Sensor Systems*, pages 387–388, New York, NY, USA, October–November 2006. ACM Press.

[26] E. Miluzzo, N.D. Lane, K. Fodor, R. Peterson, H. Lu, M. Musolesi, S.B. Eisenman, X. Zheng, and A.T. Campbell. Sensing meets mobile social networks: The design, implementation and evaluation of the cenceme application. In *The Proceedings of the 6^{th} ACM conference on Embedded network sensor systems*, pages 337–350, New York, NY, USA, November 2008. ACM.

[27] J. Moe and D.A. Carr. Understanding distributed systems via execution trace data. In *The 9^{th} International Workshop on Program Comprehension*, pages 60–70, Los Alamitos, CA, USA, May 2001. IEEE Computer Society.

[28] Moteiv Corporation. Tmote Sky datasheet.
http://www.moteiv.com/products/docs/tmote-sky-datasheet.pdf, 2006.

[29] A. Orso, J.A. Jones, and M.J. Harold. Visualization of program-execution data for deployed software. In *The 2003 ACM Symposium on Software Visualization*, pages 67–ff, New York, NY, USA, June 2003. ACM.

[30] A. Orso, J.A. Jones, M.J. Harold, and J. Stasko. GAMMATELLA: Visualization of program-execution data for deployed software. In *The 26^{th} International Conference on Software Engineering*, pages 699–700, Washington, DC, USA, May 2004. IEEE.

[31] S. Reiss and M. Renieris. Jove: Java as it happens. In *The 2005 ACM symposium on Software Visualization*, pages 115–124, New York, NY, USA, May 2005. ACM Press.

[32] S.P. Reiss. Visualizing program execution using user abstractions. In *The 2006 ACM Symposium on Software Visualization*, pages 125–134, New York, NY, USA, September 2006. ACM.

[33] Steven P. Reiss. Visualizing java in action. In *The 2003 ACM Symposium on Software visualization*, pages 57–65, New York, NY, USA, June 2003. ACM Press.

[34] J. Sajaniemi and M. Kuittinen. Program animation based on the roles of variables. In *The Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 7–15, New York, NY, USA, June 2003. ACM.

[35] J. Sallai, G. Balogh, and S. Dora. TinyDT: TinyOS plugin for the Eclipse platform. http://www.tinydt.net, October 2005. (package date).

[36] B. Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Transactions on Graphocs. Graph.*, 11(1):92–99, 1992.

[37] H.M. Sneed. Source animation as a means of program comprehension. In *The 8^{th} International Workshop on Program Comprehension*, pages 179–187, Washington, DC, USA, June 2000. IEEE.

[38] H.M. Sneed and T. Dombovari. Comprehending a complex, distributed, object-oriented software system: a report from the field. In *The 7^{th} International Workshop on Program Comprehension*, pages 218–225, Washington, DC, USA, May 1999. IEEE.

[39] J.T. Stasko. Tango: A framework and system for algorithm animation. *SIGCHI Bulletein*, 21(3):59–60, 1990.

[40] J.T. Stasko. Animating algorithms with xtango. *SIGACT News*, 23(2):67–71, 1992.