Analysis, Instrumentation, and Visualization of Embedded Network Systems: A Testbed-Based Approach

Andrew Dalton
Clemson University, adalton@clemson.edu

Follow this and additional works at: https://tigerprints.clemson.edu/all_dissertations

Part of the Computer Sciences Commons

Recommended Citation
https://tigerprints.clemson.edu/all_dissertations/237

This Dissertation is brought to you for free and open access by the Dissertations at TigerPrints. It has been accepted for inclusion in All Dissertations by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.
ANALYSIS, INSTRUMENTATION, AND VISUALIZATION OF EMBEDDED NETWORK SYSTEMS: A TESTBED-BASED APPROACH

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

by
Andrew Ryan Dalton
August 2008

Accepted by:
Dr. Jason O. Hallstrom, Committee Chair
Dr. Brian A. Malloy
Dr. Harold C. Grossman
Dr. Mark K. Smotherman
Abstract

Embedded network systems are gaining adoption and emerging as the next step in the shift toward ubiquitous computing. Deployments range in scale from tens of devices to over a thousand; applications can be massively parallel and distributed, executing over unreliable links and devices. The programming model used to develop these systems is fundamentally different than the models provided by traditional imperative programming languages; many existing software engineering tools and techniques cannot be applied. In particular, the lack of tools and techniques to analyze, instrument, and visualize these systems make their development more difficult.

We present a framework to address these difficulties in the context of the nesC development platform. The three components of this framework are: (1) a real-time interactive, open platform for analyzing program and network behavior; (2) a source code analysis and instrumentation framework to support a range of static analysis and instrumentation activities; and (3) a control-flow visualization framework for resource-constrained embedded network devices.
Dedication

For Mom
Acknowledgments

I would like to thank my family for their love and support over the years. I would especially like to thank my mother, without whom none of this would have been possible. I would also like to thank my advisor, Dr. Jason O. Hallstrom, for his support, guidance, and friendship. Special thanks goes to everyone in the Computer Science Department at Appalachian State University for preparing me for this challenge, and to everyone here in the School of Computing at Clemson for getting me through it. I would also like to thank Sally K. Wahba and Sravanthi Dandamudi for their contributions to the Visualization Testbed discussed in Chapter 5. Finally, I would like to thank the School of Computing, the College of Engineering and Science, and the Graduate School for their awards in recognition of my research here at Clemson.
# Table of Contents

Title Page.................................................. i
Abstract................................................... ii
Dedication.................................................. iii
Acknowledgments........................................... iv
List of Tables.............................................. vii
List of Figures............................................ viii
List of Listings............................................ ix

1 Introduction........................................... 1
  1.1 Problem Statement.................................... 2
  1.2 Research Approach and Contributions............... 3
  1.3 Dissertation Organization.......................... 7

2 Background............................................. 8
  2.1 nesC Language Overview............................ 8
  2.2 Example nesC Application: TsrSensing............... 11
  2.3 TsrSensing Compared to an Imperative Implementation 17

3 Interactive Testing Framework......................... 19
  3.1 System Architecture............................... 21
  3.2 System Use-Cases.................................... 27
  3.3 The NESTShell Interface............................ 37
  3.4 Evaluation........................................... 48
  3.5 Research Contributions............................ 54

4 Analysis and Instrumentation Framework............... 56
  4.1 Toolkit Implementation............................. 57
  4.2 Use-Case Scenario................................... 69
  4.3 Evaluation.......................................... 77
  4.4 Research Contributions............................. 89
## 5 Control-Flow Visualization Framework

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Visualization Toolkit</td>
<td>93</td>
</tr>
<tr>
<td>5.2 Expansion and Testbed Integration</td>
<td>100</td>
</tr>
<tr>
<td>5.3 System Use-Cases</td>
<td>103</td>
</tr>
<tr>
<td>5.4 Evaluation</td>
<td>113</td>
</tr>
<tr>
<td>5.5 Research Contributions</td>
<td>119</td>
</tr>
</tbody>
</table>

## 6 Related Work

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Program Testing, Debugging, and Profiling</td>
<td>121</td>
</tr>
<tr>
<td>6.2 Program Analysis and Instrumentation</td>
<td>126</td>
</tr>
<tr>
<td>6.3 Fault Localization and Program Visualization</td>
<td>131</td>
</tr>
</tbody>
</table>

## 7 Conclusion

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Contribution Summary</td>
<td>137</td>
</tr>
<tr>
<td>7.2 Expected Impact</td>
<td>141</td>
</tr>
</tbody>
</table>

Bibliography | 142 |
List of Tables

3.1 NESTShell Command Summary ........................................... 39
4.1 Traversal Methods (partial) ............................................. 61
4.2 Modification Methods (partial) ......................................... 64
4.3 Generation Methods (partial) ........................................... 65
4.4 Source File Parsing Times .............................................. 80
4.5 AST Traversal Times .................................................. 81
4.6 Generic Instantiation Times ........................................... 82
4.7 Interface Source File AST Sizes ..................................... 83
4.8 Configuration Source File AST Sizes ............................... 85
4.9 Module Source File AST Sizes ....................................... 86
4.10 Header File AST Sizes (nonempty) ................................. 87
5.1 Sequence Diagram Color Mapping .................................... 98
5.2 Visualization Toolkit Event Logging Time .......................... 115
5.3 Visualization Testbed Event Logging Time ....................... 116
5.4 Base Overhead (RAM/ROM) ........................................ 117
List of Figures

1.1 Overview of Contributions ............................................ 4

3.1 NESTbed System Architecture ........................................ 22
3.2 NESTbed Prototype Deployment ....................................... 23
3.3 NESTbed Server Software Architecture .............................. 24
3.4 NESTbed Manager ....................................................... 27
3.5 Deployment Configuration Manager .................................. 28
3.6 Component Rewiring Dialog .......................................... 30
3.7 Compilation Status Window .......................................... 30
3.8 Video Feeds ............................................................ 32
3.9 Mote Detail ............................................................. 34
3.10 Message Monitoring ................................................... 34
3.11 NESTShell File System Structure .................................. 38
3.12 ls Command Results .................................................. 43
3.13 Link Quality ........................................................... 51
3.14 Aggregate Network Link Quality .................................... 52
3.15 Routing Tree ........................................................... 54

4.1 Analysis and Instrumentation Process Overview .................... 58
4.2 AST Explorer Screen Capture ........................................ 68
4.3 Predicate Evaluator Screen Capture .................................. 70
4.4 AST Explorer – State Variable Declaration ........................ 73
4.5 AST Traversal Times .................................................... 81
4.6 Per-File Type Median Sizes .......................................... 88

5.1 Visualization Process Overview ...................................... 93
5.2 Call Graph for P ....................................................... 99
5.3 Testbed Visualization Process Overview ........................... 101
5.4 Call Graph Generator ................................................ 105
5.5 Probe Selector ........................................................ 105
5.6 Sequence Diagram for Blink ......................................... 106
5.7 Sequence Diagram for RadioCountToLeds ......................... 108
5.8 Generated Call Graph ................................................ 110
5.9 Visualization Filter .................................................... 111
5.10 Generated Sequence Diagrams ...................................... 113
5.11 Base Overhead ........................................................ 118
5.12 Incremental Overhead (ROM) ....................................... 119
## List of Listings

2.1 nesC SenseM Module .................................................. 12
2.2 nesC Display Interface .................................................. 14
2.3 Sample Implementation of the Display Interface .................. 15
2.4 nesC Configuration LedsDisplayC ................................. 16
2.5 Sample nesC Wiring Diagram ......................................... 17
2.6 Pseudo-Java Sense Application ....................................... 18

3.1 SurgeTelos Experimentation Script .................................. 45

4.1 Pseudocode for getNodesOfType() ................................ 63
4.2 Sample Generation Method ........................................... 66

5.1 Standard nesC Command ............................................... 95
5.2 Instrumented nesC Command ......................................... 95
Chapter 1

Introduction

Embedded network systems are emerging as a linchpin in the foundation of the ubiquitous computing vision — networked computing devices integrated transparently with the world around us. The lowest tiers of these systems are often composed of “motes” [110], inexpensive, resource-constrained devices that sense, process, and communicate environmental stimuli (e.g., light, motion, sound). The mote appellative reflects their increasingly small form-factors, which have progressed from the size of a matchbox [21], to the size of a quarter [22], to the size of a ballpoint pen tip [53]. Their small size, low-cost, and wire-free operation make it possible to deploy mote networks in a range of contexts, both indoors and outdoors, at scales that have already exceeded the 1000 node threshold [3]. These “smart dust” [110] networks are enabling an exciting class of applications, including ecological studies [54,69,79], active volcano monitoring [113], structural damage detection [14,47], wildfire prediction and tracking [33,50], disaster response [66], and intruder detection and classification [1,3], to name a few. Looking to the future, we expect an even richer class of applications to emerge as these networks become integrated into the international cyberinfrastructure.

Sensor systems are developed using various programming languages, including traditional languages such as C [5,34,48] and C++ [11]. However, the most common language for developing embedded network systems is nesC [43]. The nesC language, along with
TinyOS \cite{52,106}, an embedded operating system written in nesC, comprise the defacto standard platform for developing embedded network systems. We take this platform to represent the state-of-the-art in embedded network system development. As such, nesC and TinyOS are the focus of the research in this dissertation.

1.1 Problem Statement

Despite recent success and future promise, large-scale embedded network applications remain difficult to analyze, instrument, and visualize. The nesC programming model is fundamentally different than the models provided by traditional imperative programming languages. As such, many of the existing tools and techniques developed to aid in the aforementioned tasks are inapplicable. New techniques must be developed to aid in the development of reliable systems.

In recent years, the number of workshops, conferences, and journals dedicated to embedded network systems has grown dramatically. This, along with research investments targeting these systems from agencies such as the National Science Foundation and Microsoft Research, show an international research interest in their development. A close examination of this research and investment shows a significant focus on new software engineering techniques. Despite the impact of these trends, the applications that execute on embedded network systems remain difficult to design, build, and debug.

Why do these difficulties persist? The main issue is the lack of an integrated, general-purpose framework for analyzing, instrumenting, and visualizing code written for the nesC platform. This issue can be decomposed into several sub-issues. First, there are no suitable interactive frameworks for analyzing the runtime behavior of embedded network applications (e.g., real-time testing\footnote{To avoid confusion, we note that in this document, “testing” is used to describe the process of conducting experiments to verify the behavior of an application, not activities such as calculating code coverage and performing unit tests.}, debugging, and profiling). Second, there are no suitable general-purpose frameworks for analyzing and instrumenting nesC applications. The lack of such a framework limits the software engineering tools and tech-
niques that can target the nesC language. Finally, there are no techniques to visualize nesC control-flow. Without such tools, understanding the dynamic behavior of nesC programs is difficult.

How will an interactive testing framework make analyzing the runtime behavior of nesC programs easier? How will a software framework for analyzing and instrumenting nesC applications help make embedded network systems easier to develop? What types of visualization techniques can be developed using such a framework? How can these visualization techniques make developing embedded network systems easier? These are the questions that the work presented in this dissertation will answer.

1.2 Research Approach and Contributions

Our approach to solving these problems is three-fold. First, we present an interactive runtime analysis approach and associated framework implementation that aids in testing, debugging, and profiling applications. Second, we present an approach and associated framework implementation for analyzing and instrumenting nesC applications. (The framework implementation supports the development of the next contribution.) Finally, we present an approach and associated framework implementation for recording and visualizing application control-flow, which aids in developing and comprehending nesC applications.

Figure 1.1 summarizes our contributions, including the relationships among the elements. Elements providing services for other tools and techniques appear lower in the figure. Elements which leverage services provided by other tools and techniques appear above the tools and techniques providing those services. First is the Analysis and Instrumentation Framework. This provides services for analyzing the structure of applications and instrumenting their source code. Next is the Interactive Testing Framework. This supports the real-time analysis of applications executing on one or more physical devices. It uses services provided by the Analysis and Instrumentation Framework to instrument application source code with custom components automatically. Finally, the Control-Flow Visualiza-
tation Framework provides methods for instrumenting applications to record control-flow and to later visualize the recorded information. This framework depends on the Analysis and Instrumentation Framework in stand-alone mode, as well as the Interactive Testing Framework when used to visualize control-flow across a network. Each of these contributions is described in the next subsection.

1.2.1 Contributions

We provide the following contributions.

Contribution 1 – Interactive Testing Framework. Motes provide only a few LEDs to expose the inner-workings of their runtime behavior. Debugging and profiling messages sent to a basestation can provide more detailed information, but the additional instrumentation logic must be developed and integrated a priori. In effect, the “interesting” portions of an application (i.e., those portions that might contain correctness or performance deficiencies) must be identified and instrumented in advance. As a result, developers do not benefit from the rapid deployment and debugging cycles commonly used to improve programmer productivity in other development domains.

An interactive experimentation framework for embedded network system development addresses this issue. First, using the Analysis and Instrumentation Framework, the testing framework relies on the automated analysis and instrumentation of applications under test. Analysis activities reveal message structures used by an application and al-
allows the developer to monitor and record instances of those messages received from the
devices in real time. Analysis activities also identify the variables defined in nesC modules
and record those variable for profiling. Instrumentation activities incorporate new software
components into an application prior to it being compiled and deployed. This allows the
testing framework to, for example, perform automated component substitution. These sub-
stitutions allow users to test their applications under different evaluation scenarios without
requiring manual time-consuming, error-prone code modifications. Second, the approach
enables users to query the values of module variables across a network. This aids in debug-
ging and profiling by giving users a “window” into the state of a running system. Finally,
the approach enables users to modify variable values at runtime. This allows users to inject
artificial transient faults into their systems, allowing the user to, for instance, verify the
behavior a self-stabilizing algorithm implementation.

Some of the research questions that are addressed by this contribution include: How
can debugging and profiling data be collected from a large network of resource-constrained
devices? What types of errors can be identified by querying the state of module variables?
What types of errors can be identified by injecting faults? How can a testing framework sup-
port both novice and expert users? How can a testing framework support both interactive
and batch-based experimentation?

Contribution 2 – Analysis and Instrumentation Framework. No suitable
techniques for performing source-level analysis and instrumentation are available for nesC.
Analysis and instrumentation tasks are performed manually by the developer. This process
is tedious, time-consuming, and most important, error-prone.

The Analysis and Instrumentation Framework addresses this issue by enabling analy-
ysis and instrumentation techniques that accommodate the novel features of the nesC lan-
guage, including one-to-one and one-to-many component wirings, asynchronous events, and
tasks. The framework provides an API enabling the development of other software engi-
eering tools to support source analysis and instrumentation activities. The API enables
static analysis of application source code (e.g., construction of static system call graphs and
component relationship diagrams). The API also provides services for adding and removing instances of nesC constructs to and from applications. These services provide the ability to traverse, generate, or modify any segment of nesC source code programmatically.

Some of the research questions that are addressed by this contribution include: What should be the structure of the API? How should a nesC program be represented internally in order to perform the desired analysis and instrumentation activities? How can this internal representation be easily traversed, generated, and modified? What type of analysis activities should be supported? What type of instrumentation activities should be supported? What type and how much resources do these activities require?

Contribution 3 – Control-Flow Visualization Framework. As nesC programs become large and involve many events, developers must imagine all potential paths of control through the implementation components to understand the behavior of the application. Unlike sequential programs where a sequence of statements is executed linearly, systems for the nesC platform are event-driven. Programs consist of a set of event handlers that are fired in response to environmental stimuli (e.g., sound, timers). The possible interleavings that may occur quickly become unmanageable.

The Control-Flow Visualization Framework addresses this issue by providing the ability to record the flow of control during an application run and to visualize the information in a way that helps developers understand the application’s behavior. First, leveraging Contribution 2, the approach relies on the automatic insertion of control-flow recording code into nesC applications. This automatic code insertion eliminates the need for manual instrumentation, reducing the likelihood of introducing defects. The inserted code records runtime information to the EEPROM of the hosting device; the sequence of function\(^2\) calls and returns are captured. Recording runtime information to EEPROM is necessary to gather control-flow information from devices that are not connected to a base-station. Second, the approach provides support for downloading the recorded information to a base-station and visualizing the information as a UML sequence diagram. The visualiza-

\(^2\)The nesC language includes commands, events, tasks, and module-private functions. Throughout this dissertation, when it is unnecessary to distinguish between these types, we simply use the term “function”.

tion helps developers understand the actual execution ordering in particular system runs. Using this information, developers are able to identify unexpected control paths which may lead to incorrect program behavior. The approach makes the development of programs easier by allowing developers to identify and correct these unwanted control paths quickly.

The visualization approach is also adapted and integrated with Contribution 1 to provide network-wide control-flow visualization services. In addition to recording function control-flow, the framework also records the sending and receiving of messages over the radio. The radio stack is modified to include a unique sequence number with all messages. These sequence numbers are used to associate the sender of the corresponding message with all recipients of that message. The resulting visualization includes a set of sequence diagrams, one for each device in the network. The diagrams are “linked” together at the points where messages are sent and received, enabling users to navigate between causally-related events in the system.

Some of the research questions that are addressed by this contribution include: How will functions be uniquely identified using as little memory as possible? How can spurious functions be identified and filtered out? How will this information be stored efficiently on resource-constrained devices? What is the best way to visualize the control-flow of an individual node? What is the best way to visualize distributed control-flow data at multiple scales?

1.3 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents background material related to embedded network systems and the platforms used to program them. Chapter 3 describes the Interactive Testing Framework. Chapter 4 describes the Analysis and Instrumentation Framework. Chapter 5 describes the Control-Flow Visualization Framework. Chapter 6 follows with a review of related work in the area. Finally, Chapter 7 concludes with a summary of contributions.
Chapter 2

Background


nesC (for “Network Embedded System C”) is a dialect of the C programming language with support for components. TinyOS is an embedded operating system implemented as a set of nesC components. TinyOS and nesC have become the defacto standard development platform for developing embedded network systems. Our focus is on the problems associated with the development of nesC systems; hence we focus the background discussion on nesC.

The following sections provide an overview of nesC. Section 2.1 summarizes the language constructs provided by nesC. Section 2.2 provides an example that demonstrates the use of these constructs. Finally, Section 2.3 summarizes the unique features of nesC by comparing the example application to an implementation of the same application in a Java-like imperative language.

2.1 nesC Language Overview

The nesC programming language is fundamentally different from other imperative languages. The following subsections describe the constructs of the nesC language. First, interfaces, which are used to define inter-component communication paths, are described.
Next components and their constituent parts are discussed. Finally, the concurrency model provided by nesC is described.

2.1.1 Interfaces

An interface in nesC is used to define the inter-component calling behavior of a nesC program. Interfaces specify a bi-directional interaction between two components, known as the provider and user. The interactions are specified by two sets of functions, commands and events\(^1\). Commands are functions that a providing component must implement. Similarly, events are functions that a using component must implement. Command calls enable components to request services from other components; event signals enable components to indicate the completion of requested services and to notify components of the arrival of interrupts. The aggregation of commands and events within interfaces helps developers to understand inter-component protocols and allows the implementation of the behavior to be decoupled from its realization. Interfaces are realized as a set of signatures corresponding to the commands and events associated with a particular interface.

2.1.2 Components

nesC applications consist of a set of software components. Each component consists of a set of interfaces and an implementation. A component can both provide and use interfaces. Provided interfaces capture the functionality implemented by a component. Used interfaces capture the functionality, provided by other components, required to implement a component. This separation is similar to interfaces in C#. Components behave as singletons \([42]\) – only one instance of each component exists in a given system.

There are two types of components in nesC, modules and configurations. Modules encapsulate the implementation of commands associated with provided interfaces, as well as the implementation of events associated with used interfaces. Modules may additionally include private variables, functions, and tasks. These elements are accessible to any com-

\(^1\)We use the term “event” to refer to “event handlers” – code that is executed in response to an event.
mand, event, or function defined as part of the declaring module. (Tasks are discussed in more detail in Subsection 2.1.3.)

Configurations are used to map (or “wire”) used interfaces to components which provide those interfaces. The mapping is a one-to-many mapping; multiple providers can be mapped to a single user, and multiple users can be mapped to a single provider. Calls to commands or events associated with such interfaces fan out; calls are made to each implementation in a nondeterministic order. If a command returns a value, a special combine() function must be defined for the type being returned. This function is used to aggregate the values returned from each command or event into a single value. For instance, the combine() function for the bool type returns the conjunction of the values returned from each command.

A configuration can be either a partial or top-level configuration. Partial configurations map some components using interfaces to components providing those interfaces, while exposing other interfaces unmapped. Partial configurations are similar to partial template instantiations in C++ where some template parameters are specified while others are not. All applications are defined by a top-level configuration\(^2\). Top-level configurations leave no used interfaces unmapped.

2.1.3 Concurrency Model

The concurrency model provided by TinyOS is based on events and tasks. Events are invoked from an interrupt handler, or from a task to indicate the status of a service request. An event signaled from an interrupt handler may preempt tasks or other events. Events are designed for low-latency operations and allow the system to remain responsive. Longer-running operations are implemented as tasks. Tasks allow an application to request a function call at some future time. When a task is posted, it enters a queue maintained by the operating system. This model is similar to Linux workqueues [20]. When there are no currently executing interrupts, the operating system dequeues and executes tasks from

\(^2\)Although all applications are defined by a top-level configuration, applications can be composed of multiple top-level configurations.
the queue sequentially. Tasks do not preempt other tasks and run to completion. They are atomic with respect to other tasks, but are not atomic with respect to events originating from interrupt handlers.

Because the nesC execution model is concurrent, programs are susceptible to race conditions involving component state. The source code for an application can be divided into two parts: synchronous code, which is code that can only be reached from tasks, and asynchronous code, which is code that is reachable from at least one interrupt handler. To avoid race conditions, the following condition must hold: If a variable is modified from within asynchronous code, all accesses to the variable must occur in atomic statements. If a variable is read in asynchronous code, all writes to the variable must occur in atomic statements. The atomic keyword disables interrupts prior to the execution of the statement and re-enables interrupts once execution completes.

2.2 Example nesC Application: TsrSensing

To make these concepts more concrete, we consider an example nesC application that periodically polls a sensor and displays the sensor reading. Listing 2.1 contains the implementation of the SenseM component. The component uses four interfaces: Boot, Alarm, Read, and Display (Lines 2–5). Interfaces can be locally renamed to allow the same interface to be used more than once, or to associate meaning with its use (Lines 3 and 4). Recall that these interfaces define the commands available for use in the implementation of SenseM, as well as the events that must be defined by the component. With the exception of the last used interface, Display, each of these interfaces are provided as part of the TinyOS distribution. The Display interface is defined by the developer.

The first implemented event is `booted()` (Lines 12–14), which is part of the `Boot` interface. The event is used to notify the application that the system has been initialized and that the application is ready to run. In the implementation of the event, the `start()` command of the `Alarm` interface is called (Line 13); the constant `SAMPLING_FREQUENCY`
module SenseM {
  uses interface Boot;
  uses interface Alarm<TMilli, uint32_t> as SenseAlarm;
  uses interface Read<uint16_t> as SensorReader;
  uses interface Display;
}

implementation {
  #define SAMPLING_FREQUENCY 1000
  task void readSensor();

  event void Boot.booted() {
    call SenseAlarm.start(SAMPLING_FREQUENCY);
  }

  async event void SenseAlarm.fired() {
    post readSensor();
  }

  task void readSensor() {
    call SensorReader.read();
  }

  event void SensorReader.readDone(error_t result, uint16_t data) {
    if (SUCCESS == result) {
      call Display.displayValue(data);
    } else {
      call Display.clear();
    }
    call SenseAlarm.start(SAMPLING_FREQUENCY);
  }
}

Listing 2.1: nesC SenseM Module
(Line 8) is passed as an argument. Notice that the Alarm interface has been locally renamed to SenseAlarm; hence the command call is prefixed with SenseAlarm. Also note that this interface is parameterized by two types. The first, TMilli, represents the desired precision of the alarm, milliseconds, and the second, uint32_t, represents the number of milliseconds that must pass before the alarm fires, an unsigned 32-bit value. Notice also that the Read interface is parameterized by a type, uint16_t, representing the type of the data to be read. The start() command returns after the alarm has been scheduled. Next, the booted() event completes, returning control to the operating system. The component is “awakened” when the fired() event is executed.

After one second (1000 milliseconds), the operating system delivers the fired() event (Lines 16–18) associated with the operation started by the call to start(). Unlike the booted() event (Lines 12–14), the fired() event is declared to be async. This means that the event is delivered from interrupt context and may preempt other commands and events. The developer’s goal in this case is to read from the sensor each time the alarm fires. Because, however, the fired() event is delivered from interrupt context, it is not safe to make calls to commands. Commands not marked as async are assumed to execute outside of interrupt context. Therefore, the command invocation must be deferred to a task: The task readSensor() is posted to the operating system task queue (Line 17). The fired() event then completes, returning control to the operating system. At some later point, the operating system will dequeue the readSensor() task and execute it. At that point, control is transfered to the implementation of readSensor() (Lines 20–22). This task in turn calls the read() command on the Read interface (locally renamed to SensorReader; Line 4). The call to read() returns immediately, and the readSensor() task returns control to the operating system.

When the sensor data becomes available, the readDone() event of the Read interface (Lines 24 – 31) will be signaled to notify the application that the data is ready. The event accepts two parameters, result and data. The value of the first indicates whether data was successfully read from the sensor. It is possible, for instance, that the sensor was already in
interface Display {
    command void displayValue(uint16_t reading);
    command void clearDisplay();
}

Listing 2.2: nesC Display Interface

use and could not satisfy the request. The second parameter contains the retrieved sensor reading. After a successful reading, readDone() calls the displayValue() command of the Display interface, passing the data read from the sensor (Line 26). Otherwise, the clear() command of the Display interface is called (Line 28). Finally, the Alarm is restarted (Line 30) and the process repeats one second later.

It is useful to note that the description of this component has been in terms of its interfaces. We do not, for instance, know what type of sensor the program is reading. We also do not know how the sensor reading is being displayed. In nesC, the specification of the required behavior and the specification of the components that realize this behavior are performed separately.

Listing 2.2 shows the Display interface provided by the SenseM component (Listing 2.1; Line 3). The listing shows the displayValue() command (Listing 2.1; Line 26) and the clear() command called (Listing 2.1; Line 28). Note that because the Display interface does not define any events, the implementation of SenseM does not contain any events prefixed with Display.

Next we turn our attention to the LedsDisplayM module shown in Listing 2.3. Like SenseM, this module begins with a list of interfaces (Lines 2–4). Unlike SenseM, however, LedsDisplayM both uses (Boot and Leds; Lines 2 and 3) and provides interfaces (Display; Line 4). Boot is the same interface discussed previously. In this module, the booted() event (Lines 7–9) is used to initialize the state of the component providing the Leds interface (Line 8). The Leds interface enables LedsDisplayM to control the state of the LEDs on a mote (Lines 8, 12, and 16). Providing the Display interface makes LedsDisplayM responsible for implementing the commands defined as part of that interface, namely displayValue()
module LedsDisplayM {
  uses interface Boot;
  uses interface Leds;
  provides interface Display;
}

implementation {
  event void Boot.booted() {
    call Leds.set(0);
  }

  command void Display.displayValue(uint16_t value) {
    call Leds.set(value);
  }

  command void Display.clear() {
    call Leds.set(0);
  }
}

Listing 2.3: Sample Implementation of the Display Interface

(Lines 11–13) and clear() (Lines 15–17). The displayValue() command passes the value to be displayed to the set() command of the Leds interface (Line 12). The clear() command similarly passes the value 0 to the set() command of the Leds interface (Line 16). Recall that at this point we do not know how the functionality of the Leds interface is realized.

Listing 2.4 shows the configuration LedsDisplayC. This component associates providing components with some of the used interfaces, while leaving other interfaces unassociated. Like all components, LedsDisplayC begins with a set of interfaces. Like LedsDisplayM, LedsDisplayC uses the Boot interface (Line 2) and provides the Display interface (Line 3). However, unlike the module LedsDisplayM, the configuration LedsDisplayC maps components that use interfaces to components that provide those interfaces. The components that participate in the mappings defined by LedsDisplayC are LedsDisplayM and LedsC (Lines 6 and 7). LedsDisplayM is the component defined in Listing 2.3. LedsC is a component provided by TinyOS that controls the state of the LEDs on a mote.

Next, the component mappings are defined (Lines 9–12). The events associated with the Boot interface used by LedsDisplayC will be realized by LedsDisplayM (Line 9). The commands associated with the Display interface provided by LedsDisplayC will be realized by LedsDisplayM (Line 10). Finally, the commands associated with the Leds interface used by
configuration LedsDisplayC {
    uses interface Boot;
    provides interface Display;
}

implementation {
    components LedsDisplayM;
    components LedsC;

    Boot = LedsDisplayM.Boot;
    Display = LedsDisplayM.Display;

    LedsDisplayM.Leds -> LedsC.Leds;
}

Listing 2.4: nesC Configuration LedsDisplayC

LedsDisplayM will be realized by LedsC (Line 12). At this point, the participating component that will realize the Boot interface used by the LedsDisplayC component is still unknown. Similarly the participating component, if any, that will use the Display interface provided by the LedsDisplayC component is unknown. It is not necessary for all provided interfaces to be used.

Listing 2.5 shows the top-level configuration of the TsrSensing application. First, the set of modules used to complete the wirings are identified (Lines 4–8). Notice that this set includes the SenseM module (Line 5), locally renamed to Application, and the LedsDisplayC module (Line 6). Also note that generic components must be instantiated using the new operator (Lines 7 and 8). MainC is the “main” component in the system and is responsible for initializing the components within the system, notifying components that the system has booted, and scheduling tasks. AlarmMillC is a component that implements an alarm with millisecond granularity. HamamatsuS10871TsrC is a component that implements the TSR (Total Solar Radiation) sensing functionality found on the Telos [78] family of motes. Next, the wirings for the application are defined (Lines 10–14). First, the Boot interface used by the Application component (renamed from SenseM) is realized by the MainC component. Next, the Boot interface used by the LedsDisplayC component is also realized by the MainC component. (Recall that whenever the component mapping is one-to-many, the associated commands and events fan-out in a nondeterministic manner.) Next, the Display interface
configuration TsrSensing {
}
implementation {
    components MainC;
    components SenseM as Application;
    components LedsDisplayC;
    components new AlarmMilliC() as AlarmImpl;
    components new HamamatsuS10871TsrC() as Sensor;
    Application.Boot -> MainC.Boot;
    LedsDisplayC.Boot -> MainC.Boot;
    Application.Display -> LedsDisplayC.Display;
    Application.SenseAlarm -> AlarmImpl.AlarmMilli32;
    Application.SensorReader -> Sensor.Read;
}

Listing 2.5: Sample nesC Wiring Diagram

used by the Application component is realized by the LedsDisplayC component. Next, the SenseAlarm interface (renamed from Alarm by the SenseM module), used by the Application component is realized by the AlarmImpl component (renamed from AlarmMilliC). Finally, the SensorReader interface (renamed from Read) used by the Application component is realized by the Sensor component (renamed from HamamatsuS10871TsrC).

2.3 TsrSensing Compared to an Imperative Implementation

In contrast to the nesC example, Listing 2.6 shows what the same application might look like if implemented using an imperative Java-like language and emphasizes the novelty of the nesC language. Unlike nesC, where interface realizations are mapped to the components which use them in an external configuration component, the realization mapping occurs inline (Lines 1 and 2). Also, the Java-like implementation cannot rename components or interfaces; instead, individual variables are used to reference objects. In this example, the variables are declared within the same scope that they are used, unlike nesC programs where used interfaces are defined at the top of a component. Additionally, each method call in the Java-like implementation corresponds to exactly one method invocation. The Java-like implementation consists of only methods; there are no concepts of commands,
Display display = LedsDisplay.getInstance();
Read sensorReader = TelosLightSensor.getInstance();
for (;;) {
    display.displayValue(sensorReader.read());
    Thread.sleep(1000);
}

Listing 2.6: Pseudo-Java Sense Application

events, and tasks. The use of a Java-like interface imposes a requirement on only the callee – the interfaces are not bi-directional. These properties make the Java-like implementation significantly easier to analyze and comprehend. It is not necessary to parse and store configurations that map interface users to interface providers. It is not necessary to maintain information about component and interface renaming. It is not necessary to predict the applications behavior if multiple realizations are wired to an interface (i.e., there are no fanning function calls or returns).

Also, unlike nesC, where there are no blocking function calls, this example calls the blocking function read() (Line 4). In the presence of blocking function calls, the linear execution of the code can “wait” for long-running operations; split-phase operations are unnecessary. Blocking calls also enable repetitive operations to be included in a loop (Lines 3–6), with a blocking sleep() operation to determine the interval (Line 5) rather than being included in an event associated with a periodic alarm. There is no “yo-yo”-ing between application context and the operating system. The sequential execution model prevents the interleaving of asynchronous event handlers, which makes the program easier to understand.

There are also fewer lines of code, making the application easier to understand. The Java-like implementation is unsuitable for developing software for the resource-constrained embedded network systems that nesC targets. Such an implementation would consume more of the limited resources (e.g., RAM, ROM) available on the devices. Also, such an implementation would introduce additional latency as a result of context switches when blocking calls are executed.
Chapter 3

Interactive Testing Framework

In Chapter 1, we identified the problem that there are no suitable interactive frameworks for analyzing the runtime behavior of embedded network applications (e.g., real-time testing, debugging, and profiling). In this chapter, we describe the first of our contributions: an interactive testing framework for embedded network systems. We refer to our implementation of this framework as the Network Embedded Sensor Testbed (NESTbed). The NESTbed’s supporting middleware platform exposes multiple physical deployments, shared, in effect, as virtual devices accessible to a distributed research community. The platform is engineered to be interactive, source-centric, and open.

By interactive, we refer to a design that enables users to profile source- and network-level components in real time, as well as inject network packets and state modifications. This improves developers’ ability to evaluate system performance, localize defects, and observe behaviors in the presence of anomalous network conditions and transient faults. By source-centric, we refer to a design that targets application source materials, as opposed to application images; source-centered features include automated analysis, instrumentation, and compilation services. These services improve programmer productivity by eliminating the need for manual integration of testbed management components. Equally important, the services increase the level of implementation detail available to software testers and provide a foundation for software configuration testing. A range of application configurations can
be evaluated without the expense of developing the individual variants. Finally, by open, we refer to a design that enables developers to extend the set of interfaces used to access the testbed without modifying—or even restarting—the underlying middleware. Developers may choose to use the default graphical user interface, the default shell scripting interface, or a custom interface appropriate to a particular system or experimentation task. This chapter is based on [25,26] with supporting material from [30,31].

**Design Desiderata.** The framework design goals are as follows. First, the framework must support interactive experimentation, enabling developers to interact with executing programs in real time. Interactive experimentation will allow developers to monitor the network as their programs execute. Without interactive experimentation, developers would be required to identify the “interesting” portions of their application a priori and manually instrument the program source to gather information about its runtime behavior. Second, the framework must enable automated analysis of applications. The automated analysis must include the ability to identify message structures defined within the application. This will allow developers to select message structures for monitoring and recording. The automated analysis must also support the identification of variables defined in nesC modules. This information will allow developers to select variables to be profiled and modified. Without automated program analysis, developers would need to identify message structures and variables manually and include the information as metadata submitted with the program. Third, the framework must enable automated instrumentation of applications. Automated instrumentation will, for instance, allow the framework to integrate supporting software components into applications. Without such automated instrumentation, developers would be responsible for manually integrating such components into their applications. Finally, the framework must enable users to query and write the values of module variables within a network in real time. Querying the state of variables allows the user to collect information about the state of the running application. The ability to query variables for their values at runtime will, for example, provide developers with state information while debugging their applications. This state information will allow them to more quickly direct their attention
to the portions of their code that are not behaving as expected. Without the ability to query variables' values at runtime, when developers want to debug their applications, they must include debugging statements in their code a priori, gather debugging information from logs, use the information to guide the insertion of additional debugging logic into their program, and re-run the experiment. The ability to write to variables will, for instance, allow developers to inject artificial transient state faults into their programs (e.g., introduce a cycle in a spanning tree by modifying parent identifiers). Without the ability to modify state variables at runtime, developers would be required to include code to trigger artificial transient faults manually based on the condition of the system.

The following sections discuss the implementation and evaluation of the NESTbed system. Section 3.1 presents an overview of the NESTbed architecture, including the key hardware and software components underlying its design. Section 3.2 presents use-case scenarios that highlight key features and benefits of the system. The scenarios are presented in the context of the default graphical interface. The shell-based scripting interface is presented in Section 3.3, along with a representative example scenario. Section 3.4 presents an evaluation of the testbed in terms of the testing, debugging, and profiling activities it enables. Finally, Section 3.5 summarizes our research contributions in the area of interactive testing, debugging, and profiling embedded network systems.

3.1 System Architecture

The NESTbed system architecture is illustrated in Figure 3.1. The architecture is composed of three layers: (1) physical network deployments, (2) a centralized application and database server, and (3) client interfaces for remote users, who may optionally connect one or more remote subnets. We briefly describe each of the architectural layers in the paragraphs that follow.

The system supports multiple physical deployments. Given the design goal of enabling interactive use, each deployment is dedicated to a single user at a given time. Multiple
users may, however, access different deployments concurrently. Our prototype configuration includes one physical deployment consisting of 80 Tmote Sky [78] devices arranged in a dense grid. Small web cameras mounted overhead provide streaming video feeds that show the actuation state of the network. While the feeds are not strictly necessary for gathering this state, they appear to have an important psychological benefit. Users note that the video feeds provide an improved sense of presence; they support the view of the testbed as a locally attached virtual device.

The prototype deployment is shown in Figure 3.2. Each mote is attached to the server through a USB connection. The grid measures 4 by 8 feet. Although our current facility is not large enough to house such a deployment, the addition of wireless USB extenders would enable connections in excess of 150 feet, significantly increasing the potential deployment scale, especially if the connections were chained. No additional hardware or software changes would be required. It is natural to question whether the geographic scale of the deployment admits of interesting (and realistic) wireless topologies. We shall return to this question with an affirmative response in Section 3.2.

Each Tmote Sky includes a 16-bit microcontroller clocked at 8MHz, a 2.4GHz ZigBee radio, 48kB of ROM, 10kB of RAM, and 1MB of off-chip EEPROM storage. Each device additionally includes integrated temperature, light, and humidity sensors and can be configured to support a range of additional sensors. Each mote exposes a unique hardware identifier that enables the software running on the server to associate a physical grid position with the device, independent of how the operating system assigns port addresses.
This allows the server to preserve mote addressability across reboots. We note that the total equipment cost for the prototype installation, including the application and database server, is less than $10,000, making the deployment economically feasible to replicate at other institutions.

The back-end server hosts a suite of APIs that enable remote clients to work with NESTbed projects on one or more network deployments. Each API is implemented as a collection of Java RMI objects\(^1\) [96] referred to as “managers”. The RMI-based design simplifies the construction of remote client interfaces and exposes the testbed for programmatic control. We shall consider two client interfaces, designed and implemented independently, in Section 3.2. As illustrated in Figure 3.1, the API suite consists of six core components: (1) the Configuration API, (2) the Instrumentation and Compilation API, (3) the Deployment API, (4) the Profiling API, (5) the Power Control API, and (6) the Gateway Control API. The manager objects within each API, and the key resource dependencies among them, are illustrated in Figure 3.3. The services provided by each API are summarized below.

\(^1\)Our current API implementation uses the Java Remote Method Protocol (JRMP) for interprocess communication, requiring that client interfaces be implemented in Java. With minor modifications, the Internet Inter-Orb Protocol (IIOP) could be used [97] to enable CORBA compatibility, eliminating this restriction.
• **Configuration API.** The Configuration API consists of managers that support the construction and maintenance of NESTbed *projects* and *deployment configurations*. A project includes a collection of *nesC* source files, meta-data about the files (e.g., program symbols, message structures), and a corresponding set of deployment configurations. Each deployment configuration describes a project installation, including the mappings between application images and physical devices, network- and source-level profiling to be performed, and radio power settings for each device. Projects and configurations are maintained in a persistent store to maintain experimental controls across runs\(^2\). Data adapter objects abstract the underlying storage technology (e.g., MySQL, PostgreSQL) to enable pluggable storage implementations.

• **Instrumentation and Compilation API.** The Instrumentation and Compilation API consists of managers that support static analysis, instrumentation, and compilation of source files within a project. The analysis services include parsing functions, program symbol identification, and message structure identification. Analysis results are maintained in persistent storage to improve system response time across experiment configurations. The instrumentation services include support for integrating NESTbed management components required by the server library and integrating al-

---

\(^2\)Note, however, that some experimental controls cannot be preserved in software. External network interference, for instance, may vary from one experimental run to another.
ternative system- and application-level nesC components (e.g., alternative radio stack implementations). Component alternatives may be selected from the NESTbed library, or provided as part of a user project. The managers rely on source-level weavers to perform these functions. Note that the nesC Weaver in turn relies on the API provided by the Analysis and Instrumentation Framework (described in Chapter 4). The services provided by the Instrumentation and Compilation API contribute to achieving our design goal of providing automated application analysis and instrumentation.

- **Deployment API.** The Deployment API is implemented by a manager that provides services for programming and configuring the shared network deployments based on a specified project and deployment configuration. The services include both whole-network and individual mote programming functions, as well as error detection and reporting features. In the event of a device programming error, the client is notified of the failed installation and may optionally choose to reprogram the device.

- **Profiling API.** The Profiling API consists of managers that provide source- and network-level profiling functions. The source-level functions enable remote clients to read and write program variables associated with the application image executing on a device dynamically. The network-level functions enable clients to subscribe to one or more message streams corresponding to the network messages received by a specified network subset. Recall that the program symbols and message structures associated with each application are maintained in persistent storage. This information may be queried by remote users to assist in configuring the activities performed by the Profiling Manager. The services provided by the Profiling API contribute to achieving our design goal of enabling developers to read and write module variables’ values at runtime. Also, the services support our design goal of enabling interactive experimentation.

- **Power Control API.** The Power Control API is implemented as a manager that provides services for toggling power to specified devices. The services are implemented
using USB power control functions included as part of the *USB 2.0* \cite{19} standard. The API services enable the injection of transient and persistent node failures to support fault-tolerance experimentation. The services additionally support device recovery when nodes enter unresponsive states. The services provided by the Power Control API contribute to achieving our design goal of enabling interactive experimentation.

- **Gateway Control API.** The Gateway Control API consists of managers that enable remote users to create and destroy network gateways. This allows remote clients to extend static NESTbed deployments with remote networks, system controllers, and applications. Developers can, for instance, inject live sensor data into a NESTbed deployment from an outdoor field experiment. The API manages a set of TinyOS *SerialForwarder* instances\(^3\) that serve as mote-to-TCP bridges, one for each device. Messages received by a mote over its wireless radio are forwarded through its USB connection and retransmitted at an advertised port by the server. Messages received at a port are forwarded to the corresponding device through its USB connection and retransmitted over its wireless radio. The services provided by the Gateway Control API contribute to achieving our design goal of enabling interactive experimentation.

The NESTbed system includes two default user interfaces. The first is a graphical user-interface designed to enable “user-friendly” NESTbed access. The second is a shell-based interface that provides scripting services for complex experimentation tasks, or tasks involving a large degree of repetition. In the next sections, each interface is used to illustrate the testbed functionality summarized here. We emphasize, however, that the testbed services are exposed by the NESTbed server. Other researchers may choose to access this functionality through a custom interface. In the case of a deployment intended for continuous use, for example, researchers might choose to develop a scheduling system on top of the exposed services.

\(^3\)The *SerialForwarder* libraries are packaged as part of the TinyOS distribution \cite{106}.
3.2 System Use-Cases

We now turn our attention to a series of use-cases that illustrate the features and benefits of the NESTbed system. We begin with a scenario involving the development of a multi-hop sensing application. The scenario assumes the use of the NESTbed graphical interface.

3.2.1 The NESTbed Manager

The interface is Java-based and supports “one-click” web deployment using Java Web Start. The initial NESTbed Manager window is shown in Figure 3.4. The first display segment lists the physical network deployments available for use. In the prototype installation, the “Ultra-Dense Network” is the only available deployment. This information is configured statically by the system administrator. The second and third segments display the associated NESTbed projects and deployment configurations populated by system users. Project and configuration management functions (e.g., for adding projects, cloning configurations) are realized by the Configuration API exposed by the server.
The scenario begins with the selection of a physical network deployment and the creation of an associated NESTbed project and deployment configuration. As shown in the figure, the user has created a project named “Multi-Hop Sensing Application” associated with the “Ultra-Dense Network” and an empty configuration within the project, named “Default Configuration”. The Deployment Configuration Manager is then used to configure the system installation and associated profiling settings for a particular debugging or experimentation task.

3.2.2 The Deployment Configuration Manager

The Deployment Configuration Manager is shown in Figure 3.5. The left panel displays the nesC programs associated with the active configuration, and for each program, a list of the constituent nesC modules, module variables, and messaging structures. The panel is populated automatically as programs are uploaded by the user. The right panel displays the physical topology of the mote network; the information is configured statically.
by the system administrator. The display for each mote includes the associated network identifier, an indication of the application image to be installed, and the radio power level that should be set upon installation. The bottom panel displays a list of module variables and messaging structures selected for profiling. As we shall see, these selections control the available profiling actions when the deployment configuration is used to program the network. The NESTbed Instrumentation and Compilation API is used to generate the displayed program data.

The user scenario continues with the selection of the application source directories to be archived and uploaded to the NESTbed server. After making this selection, the user is presented with the Component Rewiring dialog shown in Figure 3.6. The dialog enables the selection of alternative operating system and application-level component implementations\(^4\). A user might, for example, upload a single source directory multiple times, selecting alternative radio stack and network routing implementations in each case\(^6\). This would eliminate the development effort normally required to construct the individual program variants and illustrates a realization of our design goal of providing automated program instrumentation. In the scenario captured in Figure 3.5, the user has uploaded the MultiHopSensing application and has chosen to use the ReliableComm component in place of the default radio stack (GenericComm). When the application was uploaded, the files were automatically parsed, instrumented, and compiled for use. The parsing and instrumentation services illustrate a realization of our design goal of enabling automated analysis and instrumentation activities. The status window shown in Figure 3.7 was displayed during the compilation process. Upon successful completion, the left panel was updated with the name of the uploaded program and its associated program symbols. MultiHopSensing de-

\(^4\)The NESTbed distribution includes two alternative radio stack implementations; these are the only operating system alternatives available for selection through the graphical interface. The system is, however, extensible to an arbitrary number and type of alternative services.

\(^5\)The alternative radio stack implementations are designed for TinyOS-1.x; they are not available for NESTbed deployments using TinyOS-2.x.

\(^6\)The Instrumentation and Compilation API assumes interface compatibility between selected components and user-provided alternatives. Syntactic errors introduced during the instrumentation process due to interface violations in user-provided components will be reported at compile time. Semantic errors cannot be checked.
fines three messaging structures and a range of program modules. The SensingM module, for instance, includes two program variables, $msg$ and $pending$, as shown in the figure.

The next step is to configure the applications to be installed on the network. This involves dragging programs from the left panel to the node(s) on which the programs should be installed. Unconfigured devices will be disabled when the configuration is activated on the network. In the ongoing scenario captured in Figure 3.5, every second device has been configured; unconfigured devices are identified by hash marks. The value shown in the bottom right corner of each mote icon indicates the radio power level to be set when the device is programmed. In this scenario, the radio power level has been reduced across the network to account for the density of the deployment. The reduction limits the effective range of each device, creating opportunities for more interesting network topologies. The goal is to generate topologies more consistent with a geographically distributed deployment. We shall return to this idea later in the section.

The final configuration step involves selecting the runtime profiling information to be made available when the configuration is activated. The user can select two types of elements from the left panel. "Module variables" selected for profiling can be inspected and modified during program execution. "Messaging structures" corresponding to the types of packets transmitted over the radio or USB port can also be selected. This enables the user to inspect the contents of messages transmitted via USB during program execution. To
enable inspection of the wireless network, the user can include simple USB forwarding logic within their programs. Alternatively, they may choose to install a radio-to-USB forwarding application on one or more of the unused motes. Several such applications are included as part of the TinyOS distribution (e.g., TOSBase) [106].

In the scenario shown in Figure 3.5, the user has selected two variables for profiling, each declared within the RoutingM module. RoutingM implements a variation of the TinyOS Beaconing Protocol [52] to maintain a shortest-path spanning tree rooted at node 0. The tree is used to route sensor data to the root node, which in turn forwards the data through its USB port for upper-tier processing. The first selected variable, distance, stores the hosting node’s distance from the root, measured in hops. The second, parent, stores the identifier of the node’s parent in the tree. Although not visible in the figure, the user has also selected the UartMsg structure, which corresponds to the messages received and forwarded by the root node. The user is now ready to activate the configuration using the NESTbed Network Monitor.

3.2.3 The Network Monitor

The interface of the Network Monitor is similar to the Deployment Configuration Manager; we omit an additional screen capture. The interface enables users to install, debug, and profile applications interactively based on the active deployment configuration. Single device and whole-network programming are supported. In the latter case, motes are programmed in parallel to reduce installation time. The former option, used less frequently, supports scenarios that require a particular installation order. For example, when evaluating fault tolerance characteristics, it may be useful to introduce corrupted nodes gradually.

Visual feedback is provided during installation to indicate success or failure. Nodes are shown within a flashing green box during installation; solid green and solid red indicate success and failure, respectively. Installation failures are often remedied by reprogramming the failed devices. It is possible, however, for a mote to enter a hardware state in which the device cannot be reprogrammed. In such a case, the device can be power-cycled to
re-enable programmability. The Deployment API is used to implement the installation and status reporting features exposed through the client interface. The Power Control API is used to implement the power-cycling feature.

Upon successful installation, users may view video feeds of the network to inspect its actuation state. These video feeds contribute to achieving our design goal of enabling interactive experimentation. Sample images are shown in Figure 3.8; each captures approximately one-half of the network (with some overlap). The actuation state consists of the LED states of the individual devices — a useful debugging tool for signaling phase transitions, error conditions, and other significant events. In general, the actuation state may also include the states of external devices under network control (e.g., lights, physical switches, motors). The goal of the feeds is to provide convenient real-time access to this information. Again, the feeds are not strictly necessary; the actuation state can be inspected through corresponding state variables. Users have noted, however, that the video feeds provide a sense of presence. They support the abstraction of the testbed as a virtual device and provide users with additional confidence in their perception of the network state.

The user scenario continues after the installation of the active configuration. Each configured mote is executing an instance of MultiHopSensing. The user is now interested
in determining whether a stable spanning tree has formed. In addition to the video feeds, the Network Monitor supports real-time debugging and profiling of individual nodes; both source- and network-level inspection are supported. After selecting the device of interest, the user is presented with the *Mote Detail* window shown in Figure 3.9.

The window summarizes information about the selected device, including its network identifier, physical location, and hardware characteristics. The bottom panel shows the variables and message structures previously selected for profiling. When a variable is selected, the system retrieves its current value and updates the display. This querying contributes to achieving our design goal of enabling users to query variables’ values in real time. In the figure, the user has queried the values of `parent` and `distance`. Because a child of node 0 is one hop from the root, the values appear correct for the selected device. The user may similarly choose to *update* a program variable. The user might choose, for instance, to inject a transient state fault to force the selection of a new parent, or set an invalid distance from the root to determine whether the system can recover from state corruption. This contributes to achieving our design goal of enabling users to modify variables’ values in real time. The static information displayed within the Mote Detail window is retrieved using the Configuration API. The variable profiling features are implemented using the Profiling API.

The user scenario continues with the task of determining whether the appropriate data packets are being received and forwarded by the root node. To achieve this, the user selects the type of message to be intercepted from the *Message Profiling* tab shown in Figure 3.9. In this case, `UartMsg` is selected, and the *Message Monitoring* window shown in Figure 3.10 is displayed. The window is generated dynamically based on the fields contained within the selected structure. In this case, the window includes a field to identify the source mote and a variety of sensor readings defined within `UartMsg`. The fields are updated in real time based on the messages transmitted over the root node’s USB port. Received messages may also be logged to the client’s local machine for later analysis. Multiple logging sessions associated with different devices may be active simultaneously. This real-
time message monitoring contributes to achieving our design goal of enabling interactive experimentation.

It is useful to consider the implementation of this feature. When an application is uploaded to the NESTbed server, its messaging structures are identified by the Instrumentation and Compilation API. A corresponding Java class is generated for each structure\(^7\). These *message classes* provide methods for parsing raw packet data and populating class fields. When a structure is selected for monitoring, the NESTbed server is notified via the Profiling API. The server in turn begins to inspect data received through the USB port of the relevant device. When a message of the appropriate type is received, the server constructs an instance of the corresponding Java data class and transmits the object to the client application. When received, the *Java Reflection API* \([95]\) is used to inspect the object and to create a Message Monitoring window of the type shown in Figure 3.10. Logged messages are recorded in Java’s *serialized object* format, simplifying the construction of external analysis tools.

\(^7\)The *Message Interface Generator* included as part of the TinyOS distribution is used to create this class \([104]\).
3.2.4 Topology Control

The spatial scale of the NESTbed deployment raises questions concerning its use in evaluating applications intended for geographically distributed environments. Office space is a factor; the current deployment measures approximately 4 by 8 feet. If space were available, however, the existing hardware components could be spaced to create a deployment in excess of 90 by 30 feet with no additional purchases or revisions. USB extenders could be used to distribute the nodes even further, potentially in excess of several hundred feet in both dimensions.

Still, the achievable scale is not without limits; there are target environments that outstrip the spatial capacity of any existing testbed. To address this limitation, the NESTbed system enables users to control the radio power level of each device. The key observation is that network link quality varies predictably as a function of transmission power and distance [109]. When a user desires a deployment environment beyond the spatial capacity of the physical network, radio power can be reduced to achieve link quality consistent with the desired distribution. The tradeoff between distance and power, and its use in emulating target environments, is also noted in [38].

It is useful to note that some scenarios cannot be faithfully emulated using this approach; the desired packet reception rate may be too low to achieve in a dense deployment, even at the lowest radio power level. In such a case, a user may substitute UniformLossyComm, included as part of the NESTbed system, in place of the default radio stack on one or more devices. (See the Component Rewiring dialog shown in Figure 3.6.) The alternative radio stack discards packets with a specified uniform probability, allowing users to emulate low-quality links consistent with large-scale spatial distribution.

3.2.5 Remote Extensions

For some experimentation tasks, a fixed indoor deployment may be insufficient; users may wish to add remote subnets, system controllers, and applications. In testing a hierarchical system, for example, it may be useful to attach a tier of computationally
rich sensor nodes that interact with a lower-tier NESTbed deployment. A user might also wish to connect a remote field deployment to inject live (or prerecorded) sensor data into a NESTbed experiment, as in [38]. External applications that analyze network performance or sensor stream data may also be required.

To enable these extensions, we adopt a variation of the *serial forwarding* approach described in [114]. When a user wishes to connect a remote service, a *network gateway* is associated with one or more NESTbed devices through the NESTbed Network Monitor. Each mote is assigned an advertised TCP port; the gateway creation function establishes a mote-to-TCP bridge on this port. More precisely, the Serial Forwarder Control API constructs a set of `SerialForwarder` instances on the NESTbed server. Each instance relays messages received through the USB port of its associated device to the corresponding TCP port and vice-versa. Remote applications connect to these ports to interact with NESTbed devices. To attach a remote subnet, a *client-side* `SerialForwarder` instance is also required; a TCP-to-TCP relay forwards packets between client- and server-side `SerialForwarder` instances. Several such relay applications are freely available.

3.2.6 Experimental Repeatability

The NESTbed system enables users to save, modify, and clone deployment configurations quickly. These features assist in improving *experimental repeatability and control*. Users can quickly retrieve and redeploy previous experiments without any risk of modifying non-environmental experimental parameters. This is especially useful in trying to replicate experimental results among NESTbed users. The ability to clone projects and configurations is also useful, improving control of variation across experiments. A user can clone a deployment configuration, make a single change, and quickly redeploy the new experiment without any risk of modifying other non-environmental parameters.
3.3 The NESTShell Interface

The graphical user interface can be cumbersome for complex multi-phase experimentation tasks and tasks involving a high degree of repetition. The open design of the NESTbed system allows end-users to provide supplementary interfaces to address these scenarios. One such interface is NESTShell, a shell-based scripting interface. In addition to supporting automation of complex and repetitive tasks, the scripting language provides constructs for interacting with client-side tools, enabling users to extend the interface as appropriate to particular scenarios.

The NESTShell interface is designed to enable remote users to interact with the NESTbed system in a manner that parallels the way in which users interact with a typical operating system shell. The goal is to provide convenient manual and script-based access to the NESTbed system features, while reducing interaction latency (by avoiding network-intensive graphics) — this, of course, without reducing the level of information detail available to end-users. At the core of the NESTShell implementation is a file system abstraction that models the hierarchical structure of (1) physical network deployments, (2) NESTbed projects, (3) deployment configurations, (4) programs, and (5) profiling data. Users navigate the file system and interact with the elements that it contains using familiar UNIX-style concepts and command primitives.

Each directory within the file system defines a command context. A user’s active directory defines the active context and dictates the set of available commands. For example, when the active directory is the project management directory, the shell provides commands for managing projects. Similarly, when the active directory is the symbol profiling directory for a particular device, the shell provides commands for reading and writing program variables defined by the application executing on the device. A directory may also contain files used to convey information about the active context. A mote directory, for instance, includes a file that specifies information about the corresponding network node.
Figure 3.11: NESTShell File System Structure

(e.g., deployment coordinates, executing program image, hardware characteristics). The contents of this file (and others) are read using standard UNIX-style commands (e.g., cat).

The NESTShell file system structure is shown in Figure 3.11. Boldface labels correspond to literals; italicized labels are place holders for names that vary. The commands applicable in each directory appear in Table 3.1. In the paragraphs that follow, we describe the purpose of each directory and the usage of the associated commands.

### 3.3.1 Experiment Configuration

The root directory of the file system contains subdirectories corresponding to the physical deployments available for use. These subdirectories are created automatically based on static configuration data exposed through the NESTbed Configuration API. Within a deployment directory, as in all directories, users have access to the standard commands. In addition, they have access to commands used to create and remove project directories.
<table>
<thead>
<tr>
<th>Command</th>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td></td>
<td>Display the contents of a file</td>
</tr>
<tr>
<td>cd</td>
<td></td>
<td>Change the working directory</td>
</tr>
<tr>
<td>ls</td>
<td></td>
<td>List the contents of the current directory</td>
</tr>
<tr>
<td>man (help)</td>
<td></td>
<td>Get help on the commands in the current directory</td>
</tr>
<tr>
<td>quit (exit)</td>
<td></td>
<td>Exit the application</td>
</tr>
<tr>
<td>alias</td>
<td></td>
<td>Create an alternate name for a command</td>
</tr>
<tr>
<td>pwd</td>
<td></td>
<td>Print the path of the working directory</td>
</tr>
<tr>
<td>set</td>
<td></td>
<td>Set the value of a variable</td>
</tr>
<tr>
<td>unset</td>
<td></td>
<td>Unset the value of a variable</td>
</tr>
<tr>
<td>echo</td>
<td></td>
<td>Display a line of text</td>
</tr>
<tr>
<td>env</td>
<td></td>
<td>Display the name and value of all variables</td>
</tr>
<tr>
<td>shell</td>
<td></td>
<td>Execute a system-level command</td>
</tr>
<tr>
<td>foreach</td>
<td></td>
<td>Loop over a list of items and execute a set of commands</td>
</tr>
<tr>
<td>iferror</td>
<td></td>
<td>Conditionally execute a set of commands if the last command failed</td>
</tr>
<tr>
<td>mkproj</td>
<td>Physical Deployment</td>
<td>Create a new project by name</td>
</tr>
<tr>
<td>rmproj</td>
<td></td>
<td>Remove an existing project by name</td>
</tr>
<tr>
<td>mkconf</td>
<td>Project</td>
<td>Create a new deployment configuration by name</td>
</tr>
<tr>
<td>rmconf</td>
<td></td>
<td>Remove an existing deployment configuration by name</td>
</tr>
<tr>
<td>upload</td>
<td>Programs</td>
<td>Upload a new program</td>
</tr>
<tr>
<td>rm</td>
<td></td>
<td>Remove an existing program</td>
</tr>
<tr>
<td>profile</td>
<td>Messages Symbols/ Module</td>
<td>Select a message type to be profiled</td>
</tr>
<tr>
<td>rm</td>
<td>SymbolProfiling</td>
<td>Select a program symbol to be profiled</td>
</tr>
<tr>
<td>configure</td>
<td>Motes</td>
<td>Deselect a program symbol to be profiled</td>
</tr>
<tr>
<td>unconfigure</td>
<td></td>
<td>Deselect a message type to be profiled</td>
</tr>
<tr>
<td>ls</td>
<td>Motes</td>
<td>Configure a mote to run a program at the specified radio power level</td>
</tr>
<tr>
<td>install</td>
<td>Network Monitor</td>
<td>Unconfigure the specified mote</td>
</tr>
<tr>
<td>wait</td>
<td></td>
<td>Directory-specific ls; displays network information</td>
</tr>
<tr>
<td>reset</td>
<td></td>
<td>Directory-specific ls; displays network information and mote state</td>
</tr>
<tr>
<td>powerOn</td>
<td>Network Monitor</td>
<td>Installs a program on the specified mote</td>
</tr>
<tr>
<td>powerOff</td>
<td></td>
<td>Wait for current installations to complete</td>
</tr>
<tr>
<td>mkgw</td>
<td></td>
<td>Perform a soft reset on the specified mote</td>
</tr>
<tr>
<td>rmgw</td>
<td></td>
<td>Power-on the specified mote</td>
</tr>
<tr>
<td>query</td>
<td>Mote/SymbolProfiling</td>
<td>Power-off the specified mote</td>
</tr>
<tr>
<td>write</td>
<td></td>
<td>Create a network gateway associated with the specified mote</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Destroy a network gateway associated with the specified mote</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Query the mote for the value of the specified symbol</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write the specified value to the specified symbol</td>
</tr>
</tbody>
</table>

Table 3.1: NESTShell Command Summary
The project creation command requires project name and description arguments. The description is stored within a file located in the corresponding directory.

Deployment configurations are represented as subdirectories beneath each project and are managed in a similar manner. As shown in the figure, a configuration directory contains five subdirectories: (1) Programs, (2) SymbolProfiling, (3) MessageProfiling, (4) Motes, and (5) NetworkMonitor. We describe each of these directories and the subdirectories they contain (in a depth-first fashion) in the paragraphs that follow.

As its name suggests, Programs contains subdirectories corresponding to the applications uploaded by an end-user. The associated command context includes commands for uploading and removing applications. The upload command requires an application name as argument, an associated description, and a path to the application source materials on the user’s local machine. The command is implemented using the NESTbed Configuration API and the Instrumentation and Compilation API. When the command completes (and the application data has been uploaded to the server), the new program directory is created (beneath Programs), and two subdirectories are created beneath it, Symbols and Messages. The first subdirectory corresponds to program symbols and contains subdirectories that match the nesC modules defined within the uploaded application. Within each of these subdirectories are files corresponding to the program symbols declared by the respective module. The associated command context enables users to select a program symbol for profiling, making it available for runtime access. The Messages subdirectory contains files corresponding to the message structures declared by the uploaded application. These structures are not associated with particular modules — hence the omission of the module directories. The associated command context is analogous to that associated with the Symbols directory.

The next subdirectories beneath a deployment configuration are SymbolProfiling and MessageProfiling. These contain files corresponding to the program symbols and message

---

8Applications uploaded by a user are shared across deployment configurations within a project. Hence, although each configuration directory includes a Programs subdirectory, this is only a syntactic convenience; the Programs subdirectory is conceptually stored beneath the containing project directory.
structures, respectively, that have been selected for runtime profiling. The files are populated based on the selection commands discussed in the preceding paragraph. The associated command contexts include commands to de-select symbols and messages to cancel profiling of previously selected elements.

Next is the Motes directory, which provides a context for configuring the images to install on the network when the current deployment configuration is activated. The directory contains a file for each mote which specifies information about the hardware characteristics of the device (e.g., network address, deployment coordinates, memory capacity) and its current configuration status. The configuration status includes the application to install on the device and the radio power level to be set when the application is executed. The command context for the Motes directory overrides the standard ls command to provide a formatted display that includes the configuration status of each mote. A sample of the output produced by this command is shown in Figure 3.12a. The numbers in parentheses indicate the radio power level of configured motes; unconfigured motes are shown in square brackets. Additional detail (e.g., program image information, hardware characteristics) can be retrieved by invoking cat on the individual mote files. This information is retrieved using the NESTbed Configuration API. The context additionally provides commands for configuring and unconfiguring a device. The configuration command requires the address of a device, the name of an application contained in the Programs directory, and the desired radio power level. The command used to unconfigure a device clears the configuration status of the mote specified as argument. The commands to configure and unconfigure a device are implemented using the NESTbed Configuration API.

3.3.2 Experiment Execution

Each deployment configuration directory includes a NetworkMonitor subdirectory that defines a context for controlling the current network deployment. The most important commands provided in this context are install and wait. The install command is used to activate the mote configuration associated with the selected deployment configuration. This
involves programming the device using the application image mapped to it in the Motes directory and setting the requested power level upon installation. The command executes asynchronously to allow users to initiate concurrent installation requests — to, for example, activate the current deployment configuration in a whole-network fashion. After initiating a sequence of install commands, the user can issue a wait command to block until the programming step completes; the user will be notified of the aggregate installation results upon completion. The context includes additional commands to perform a soft reset on a specified mote and to toggle the power supply to a specified mote. Finally, the context provides commands to create and destroy SerialForwarder gateways. The gateway creation command requires the address of the device that will serve as the gateway and prints the resulting server-assigned IP port. The command used to destroy a gateway accepts a mote address as argument and frees the resources associated with the corresponding gateway. The NESTbed Deployment API is used to implement the installation features exposed through the NESTShell. The Power Control API is used to implement the reset and power cycling features. The Gateway Control API is used to manage gateways.

NetworkMonitor includes subdirectories corresponding to the network nodes. As before, each subdirectory contains a file that specifies information about the hardware characteristics of the corresponding device. In addition, each file specifies information about the device activity state. Initially, each device is in an unknown state because the runtime state of the network is not maintained in persistent storage. When an install command is issued on a device, the device state changes to installing. Depending on whether the program installation succeeds, the mote enters either the programmed state or the failed state. When a device is in the programmed state, it can be used as a gateway, at which point it enters the gateway state. (When the gateway is destroyed, the device returns to the programmed state.) To simplify the collection of aggregate status information, the command context of the NetworkMonitor directory overrides the standard ls command to include information about the activity state of the network. A sample of the output produced by this command

42
is shown in Figure 3.12b. The symbol shown in brackets indicates the respective node’s activity state (i.e., \( P = \text{programmed} \), \( U = \text{unknown} \)).

Finally, each mote directory beneath NetworkMonitor includes two subdirectories used for profiling purposes, ProfilingSymbols and ProfilingMessages. ProfilingSymbols contains files corresponding to the program symbols previously selected for profiling; the file names match those contained in SymbolProfiling. Each file contains the most recent value recorded for the corresponding symbol. The command context includes a query command to update this value based on the symbol’s current runtime value. It additionally includes a write command to overwrite the existing value, which in turn causes the state of the executing device to be modified. The ProfilingMessages directory is defined analogously; its contents mirror those of MessageProfiling. The command context for ProfilingMessages includes commands to subscribe and unsubscribe to message streams. When a user subscribes to a message stream associated with a particular message structure, the content of the corresponding file is initially cleared. Messages received over the USB port of the active device that are of the appropriate message type are appended to this file. Each log entry includes a line-separated list of the values contained within the record fields using a simple field=value format. The logging process continues until the user unsubscribes from the message stream. The variable querying and modification features and the message profiling features are implemented using the NESTbed Profiling API.
3.3.3 Environment Variables and Control Flow

The NESTShell interface includes a global *environment map* used to store variables that can be referenced in NESTShell commands. The commands used to interact with the environment are similar to those found in UNIX-style operating system shells. The commands `set`, `unset`, and `env`, for example, allow a user to set the value of an environment variable, remove a variable, and display the contents of the environment, respectively. The familiar `${variable}` notation is used to access the value of a variable. As in an operating system shell, the environment map simplifies the interface by enabling users to define aliases for complex or recurring strings. We shall see in Subsection 3.3.4 that this is especially useful in the case of NESTShell scripts, where environment variables serve as a convenient parameterization mechanism.

In addition to user-defined variables, the environment contains the status *system* variable. This variable stores the *exit status* of the last executed command. It might, for example, be used to determine whether a write against a particular program symbol was successful, and to trigger the execution of associated recovery logic if it was not.

We note that the interface also includes *iteration* and *conditional evaluation* constructs to enable users to express more complex experimentation and evaluation scenarios. We shall see examples of these constructs in Subsection 3.3.4.

3.3.4 Example Script

To illustrate the use of the NESTShell interface and the experimentation scenarios it enables, we consider a simple example. Listing 3.1 includes a portion of a script used to collect profiling information from a modified version of *SurgeTelos*, a spanning-tree-based, multi-hop sensing application included as part of the TinyOS distribution. For the sake of presentation, we demonstrate the interface using an *experimentation script* that can be executed by the shell interpreter. Alternatively, the contents of the script can be entered interactively. The NESTShell in interactive mode contributes to achieving our design goal of enabling interactive experimentation.
set MOTES="[0-79]"
set LEVELS="[1-3]"
cd "Ultra-Dense Network"; cd "Surge Evaluation"
iferror; then
  mkproj "Surge Evaluation" "Evaluation of RSSI/LQI"
  cd "Surge Evaluation"
endif
foreach powerLevel in ${LEVELS} do
  mkconf "Power Level ${powerLevel}" "SurgeTelos at Power Level ${powerLevel}"
done
cd "Power Level 1"; cd Programs
upload SurgeTelos "SurgeTelos Application" /opt/tinyos-1.x/apps/SurgeTelos
... iferror, exit ...
foreach powerLevel in ${LEVELS} do
  cd /
  cd "Ultra-Dense Network"
  cd "Surge Evaluation"
  cd "Power Level ${powerLevel}"
  cd Programs; cd SurgeTelos; cd Symbols
  cd MultiHopLQI
  foreach i in rawRSSI rawLQI gbCurrentParent gbCurrentHopCount gbCurrentLinkEst do
    profile ${i}
done
... cd ..; cd .. back to configuration directory ...
foreach i in ${MOTES} do
  configure ${i} SurgeTelos ${powerLevel}
done
cd ..; cd NetworkMonitor
foreach i in ${MOTES} do
  install ${i}
done
echo "Waiting for installation to complete"
wait
... iferror, exit ...
echo "Waiting for experiment to complete"
shell sleep 1m
foreach mote in ${MOTES} do
  echo "Querying ${mote} symbols"
  cd ${mote}; cd ProfilingSymbols
  foreach sym in rawRSSI rawLQI gbCurrentParent gbCurrentHopCount gbCurrentLinkEst do
    query MultiHopLQI.${sym}
done
... cd ..; cd .. ...

done
done

Listing 3.1: SurgeTelos Experimentation Script
SurgeTelos implements a distributed sensing infrastructure. Participating nodes execute a spanning tree protocol, with a preselected mote serving as the root node. Each device periodically polls its attached photo sensor and conveys the readings to the root node using the spanning tree as a routing structure. We modified the basic application to record the RSSI (received signal strength) and LQI (link quality indicator) readings associated with the last packet received from each node’s parent in the spanning tree. RSSI and LQI readings are commonly used as link quality metrics and inform the parent selection process in the SurgeTelos implementation.

The profiling task involves installing the SurgeTelos application under three different deployment configurations, each corresponding to a different radio power setting. In each configuration, the goal is to allow the routing tree to stabilize for a period of time before collecting five elements of profiling data from each node: (1) the RSSI and (2) the LQI readings mentioned previously, (3) the address of the node’s parent, (4) the node’s hop count from the root, and (5) the internal link quality metric used to inform parent selection.

The experimentation script used to perform the evaluation task appears in Listing 3.1. Key elements are described in the paragraphs that follow.

**Lines 1–2.** The script first declares the variables MOTES and LEVELS, used to parameterize the subset of devices to be programmed and the power levels to be considered, respectively. This enables users to modify the script easily to execute on the network subset and power levels of interest.

**Lines 3–7.** Next, the “Surge Evaluation” project is selected within the “Ultra-Dense Network” deployment. The iferror condition checks the value of the status variable (set by each NESTShell command) to determine whether the selection was successful. Hence, if the project does not exist, it will be created. The second parameter to mkproj provides a description for the new project. At the termination of the block, the current directory is set to the new project directory.

**Lines 8–10.** Within the project directory, each of the three deployment configurations are created. This is achieved using a foreach construct that iterates through each
of the power levels defined in LEVELS. The mkconf command is analogous to the mkproj command; note, however, the use of the powerLevel variable in defining the name of the deployment configuration directory and the associated description.

**Lines 11–13.** Next, the current directory is changed to the Programs subdirectory beneath the first configuration. The SurgeTelos application is then uploaded from the user’s local machine, using the specified name and description. (Recall that uploaded applications are shared across the deployment configurations within a project.)

**Line 14.** The remainder of the script is contained within the body of the loop initiated on this line. It iterates through the selected power levels to (1) complete the process of configuring each deployment configuration, (2) activate each configuration, and (3) collect the required profiling data.

**Lines 15–26.** The first step within the loop body is to select the program symbols to be profiled. This is achieved by changing the current directory to the MultiHopLQI module directory. The nesC module of the same name defines the symbols of interest. These symbols, selected in the body of the foreach loop, correspond to the five data elements enumerated in the discussion of our profiling goals.

**Lines 27–31.** In the Motes directory, the foreach block configures each device in the selected subset. Given the value of the MOTES variable, all 80 motes are configured with the SurgeTelos application at the current power level.

**Lines 32–38.** Next, the install command is used to activate the current deployment configuration on each device in the network. Recall that this command executes asynchronously; the network is programmed in parallel. The wait command blocks until the pending installs are complete and sets the status variable (used by iferror) appropriately.

**Lines 39–40.** When the installation process completes, the experimentation script remains idle for one minute to allow the network routing structure to stabilize. Note that the shell command enables a NESTShell script to invoke commands in the hosting operating system shell. In this case, the UNIX sleep command is used to implement the idle period.
Finally, after the one minute idle period, the script iterates through each device and queries the runtime value of each of the five program symbols being profiled. Note that in addition to updating the content of the relevant symbol file, the query command displays the retrieved value to the console. (If desired, the script output can be redirected to a file using standard redirection primitives.)

3.3.5 Interface Interoperability

We conclude this section by emphasizing that the NESTShell interface is intended to complement the default graphical interface. In some scenarios, the graphical interface is appropriate; in others, the NESTShell interface is a better choice. A novice user might, for example, prefer using the testbed through a “point-and-click” interface for simple debugging tasks. An expert user performing a series of complex experiments is likely to prefer the shell-based interface. The point is that the user is free to choose the interface that best addresses the task at hand.

Finally, we note that there are some features provided by the graphical user interface that are not provided by the NESTShell interface. In particular, the latter interface does not provide commands for user-selected component substitution. The manner in which these features should be integrated with the shell abstraction is unclear. As a stop-gap measure, users can access the instrumentation features through the graphical interface as part of configuring a NESTbed project. This same project may then be accessed using the NESTShell interface.

3.4 Evaluation

The NESTbed provides a testbed enabling the real-time testing, debugging, and profiling of embedded network systems. In this section, we evaluate the testbed in terms of its efficacy by providing representative examples of how it has been used to perform
these activities at Clemson University and Cleveland State University. Our evaluation of the testbed is based on the following objectives. The testbed must:

- **O1.1** Aid in debugging embedded network systems
- **O1.2** Aid in testing embedded network systems
- **O1.3** Enable performance analyses of embedded network systems
- **O1.4** Enable runtime profiling of embedded network systems

The following subsections describe two relevant projects enabled by the NESTbed system. Subsection 3.4.1 evaluates the testbed’s support for testing and debugging distributed applications, as well as its support for application profiling. Subsection 3.4.2 evaluates the testbed’s support for application profiling.

### 3.4.1 Reliable Communication

Recall that a user may substitute an alternative radio stack implementation as part of the deployment configuration process. Section 3.2 highlighted ReliableComm, an alternative implementation designed to improve the reliability of wireless network links. Before its inclusion as part of the NESTbed distribution, the implementation was developed, debugged, and evaluated using the NESTbed system.

When testing ReliableComm using the NESTbed system, we found that some nodes exhibited unexpected behaviors under high load. First, and most obviously, the faulty nodes exhibited unexpected LED states. Second, by observing the values of program variables at key system execution points, we were able to determine that certain program invariants had been violated. The runtime observations made possible by the NESTbed design ultimately led us to discover a synchronization error in one of the interrupt-driven state machines. Based on our past experiences debugging similar errors, we believe the NESTbed design allowed us to correct the error in a small fraction of the time it would otherwise have taken. The runtime network- and program-level visibility the design affords is an important
debugging aid, especially in debugging problems that are difficult to replicate. Concurrency and memory-related errors, for example, are notoriously difficult to reproduce — and these are exactly the types of errors repeatedly encountered when developing embedded network systems. This example illustrates the NESTbed’s utility in aiding in debugging and testing embedded network systems (O1.1 and O1.2).

To evaluate the performance of ReliableComm, we used the NESTbed interface to install a test program on each mote. The program instructs the host device to transmit packets to each of its neighbors at a specified rate for a specified duration. Each mote also records the number of messages received from each of its neighbors. By dividing the number of messages received on each link by the number of messages transmitted on the link, we are able to calculate the PRR on the link.

To control the experiment, we developed a Java application to communicate with a designated leader node through its USB port. To enable connectivity from a remote location, we used the NESTbed interface to construct a network gateway and then connected the Java application to this gateway. Upon termination of the experiment, we collected the results through the same gateway. The application was tested using both GenericComm and ReliableComm to compare the relative performance of the radio stack implementations. The NESTbed rewiring interface was used to perform this configuration step, eliminating the need for additional programming. A portion of the performance results collected using the NESTbed system are shown in Figures 3.13 and 3.14. We note that these figures illustrate the types of performance data that can be collected using the NESTbed.

Figure 3.13a presents a graphical representation of link quality, as measured by the test application deployed with GenericComm. Figure 3.13b is analogous and corresponds to ReliableComm. Each column represents a transmitting node, and each row represents a receiving node; the shading of the cell at their intersection represents the PRR of the link. The cells are shaded on a uniform scale from black to white, with black denoting a PRR of 0 percent, and white denoting a PRR of 100 percent. The aggregate impact on link quality is illustrated by the histogram shown in Figure 3.14. There are 10 link categories.
Figure 3.13: Link Quality

considered; vertical bars represent the number of links in each category. By examining the results captured in these figures —results made possible by the NESTbed system—it becomes immediately clear that ReliableComm significantly improves the reliability of mid-quality links without negatively impacting high-quality links. Packets destined for low-quality links are silently discarded to reduce network congestion. This example shows that the NESTbed system enables developers to analyze the performance of embedded network systems (O1.3).

The implementation of ReliableComm is interesting in its own right and serves as a useful radio stack alternative for NESTbed users. However, we summarize our development and evaluation experiences here only to emphasize the types of remote evaluation studies made possible by the NESTbed design.

3.4.2 Student Experimentation

In addition to its use as a research instrument, the NESTbed system is a valuable teaching tool. At Clemson the system is used as part of a graduate course in embedded
network system design. The course covers both algorithmic issues and software engineering principles as they relate to the development of large-scale embedded deployments.

Given the emphasis on large networks, scalability is a major theme of the course. In the first offering, before the NESTbed system was available for use, scalability issues were difficult to motivate, and even more difficult to analyze. Each student had access to only a small number of motes (i.e., 5–10) on which to perform their assignments. As a result, they were unable to gain experience addressing realistic congestion problems, hardware load limitations, synchronization defects, and other difficulties magnified in large-scale networks. Assignment solutions lacking required scalability properties might appear correct because they could not be tested at scale. Further, with only a few LEDs (and perhaps a few preprogrammed debugging messages) to expose the inner-workings of an algorithm implementation, it was difficult for students to evaluate implementation correctness and performance.

In the second offering of the course, students were additionally given access to the NESTbed system. Consider as an example, the second major assignment, for which students were required to demonstrate the correctness of their solutions on a minimum of 35 nodes. The assignment required the development of a multi-hop sensing application that
closely parallels the scenario presented in Section 3.2. The most challenging aspect of the assignment was the development of a self-stabilizing routing tree that tolerates multiple node failures.

The NESTbed system made it easy for students to install their applications on a large number of nodes quickly and to configure the radio power level of each device to ensure the construction of interesting routing topologies. During each demonstration, the system was used to query the values of program variables on a subset of the network. More specifically, the routing tree component was examined to determine the parent identifier of each node, as well as the distance of each node from the root of the tree. By inspecting the variables that store this information at various points in the network, it was easy to construct quick maps of the routing tree. This illustrates the NESTbed’s utility in enabling runtime profiling of embedded network systems (O1.4). One such routing tree, constructed using information gleaned through profiling on a slightly larger network, is shown in Figure 3.15a.

Each circle in the figure represents a network node; nodes selected for profiling are shown in boldface. The identifier of each node is shown at the top of its corresponding circle. The pair of numbers at the bottom of each circle represent the values retrieved during profiling – the identifier of the node’s parent and its distance from the root, respectively. The arrows between each node depict the routing links implied by these values. (Note that even in this spatially constrained deployment, the NESTbed power management features enable interesting network topologies.) The rapid development of such a map, made possible by the NESTbed system, provides immediate student feedback concerning the correctness of their routing implementation.

After demonstrating the initial formation of the routing tree, students were required to demonstrate tolerance to node failure. To achieve this, power was cut to a subset of the network nodes, causing immediate fail-stop faults. The query system was again used to construct a map of the new routing tree, and illustrates how the system aids in testing and enables runtime profiling of embedded network systems (O1.2 and O1.4). Figure 3.15b shows the routing map constructed immediately following a fail-stop fault involving node 4,
marked with an “X” in the figure. Arrows with solid heads represent routing links unaffected by the fault. Arrows with hollow heads represent affected links. Again, this type of map provides immediate feedback that would otherwise be time-consuming and error-prone to collect.

3.5 Research Contributions

We have described the design and implementation of the NESTbed, our implementation of the Interactive Testing Framework — Contribution 1 of this dissertation. We identified design desiderata and discussed how our implementation satisfies the desiderata. We showed how our *open* architecture makes the testbed extensible, enabling various client-side implementations to be developed without modifying the core implementation. We presented two such client-side implementations, a graphical application and a shell-based, command-line application. Using example scenarios, we showed how these applications can be used to collect debugging and profiling data from a network interactively. Finally, we identified four evaluation objectives and showed how the implementation of the NESTbed satisfies each of those objectives.
Prior to this work, there were no suitable frameworks that enabled developers to test, debug, and profile embedded network systems. Developers were forced to use motes’ LEDs to glean information about the inner-workings of their applications. As a result, they were unable to benefit from the rapid maintenance, deployment, and debugging cycles commonly used to improve programmer productivity in other development domains.

The Interactive Testing Framework solves these problems. First, the approach enables automated analysis and instrumentation of applications. Analysis activities enable the framework to extract static information regarding the message structures and variables defined within the system automatically. Instrumentation activities make it possible for the framework to support automatic component insertion and substitution. Second, the approach enables users to query variables’ values interactively at runtime. This provides a “window” into the running system, allowing developers to more easily debug and profile their systems. Finally, the approach enables users to write variables’ values interactively at runtime. This allows users to further test their applications by introducing artificial transient state faults into their programs. This contribution has, and will continue to improve the development of embedded network systems.
Chapter 4

Analysis and Instrumentation Framework

In Chapter 1, we identified the problem that there are no suitable general-purpose frameworks for analyzing and instrumenting nesC applications. In this chapter, we describe our second contribution – an analysis and instrumentation framework for the nesC language. The framework provides a foundation for extending automated software engineering methods to the domain of network embedded systems. It provides an API that enables users to (1) build in-memory representations of nesC programs, (2) traverse and modify the representations, (3) generate portions of the representations, and (4) regenerate nesC source materials from the representations. The framework also includes an application that enables users to visualize the representations. These visualizations serve as a guide while users develop new software engineering tools using the API and provides a point of reference for their traversal, modification, and generation tasks. We refer to our implementation of this framework as the nesC Analysis and Instrumentation Toolkit (nAIT; pronounced “nate”). This chapter is based on [27].

Design Desiderata. The framework design goals are as follows. First, the framework must enable users to construct new software engineering tools that operate on programs written for any hardware or simulator platform capable of running nesC applications.
Cross-platform support will enable users to develop generic tools that can be applied to programs written for any mote platform. Second, the toolkit must enable users to develop these tools easily. The second goal can be divided into three sub-goals. The toolkit must enable users to easily: (1) traverse the in-memory representation of programs, (2) modify existing programs, and (3) generate new programs and program segments. These features will make the code required to realize new software engineering tools shorter, thereby simplifying their development. Finally, the toolkit must normalize the in-memory representations by reducing the inherent syntactic variation found in nesC source files. This normalization will simplify code traversal and modification tasks by allowing developers to build their tools to handle a small subset of the possible syntactic variations. The lack of such a framework has, until now, restricted the development of these new software engineering tools for the nesC platform.

The following sections discuss the implementation and evaluation of nAIT, our implementation of the Analysis and Instrumentation Framework. Section 4.1 presents an overview of the toolkit’s implementation. Section 4.2 presents a representative use-case scenario. (Our third contribution, detailed in Chapter 5, also serves as a use-case for the toolkit.) Section 4.3 presents an evaluation of the toolkit in terms of its suitability of purpose, the time required to load, traverse, and modify the in-memory representation of nesC programs, as well as the associated memory overhead. Finally, Section 4.4 summaries our research contributions in the area of embedded network system analysis and instrumentation.

4.1 Toolkit Implementation

The steps involved in applying nAIT to a nesC application are summarized in Figure 4.1. In the figure, solid arrow heads represent data flow through the system, while open arrow heads represent client (e.g., users, software engineering tools) interactions with the system. The first step in applying the toolkit is to scan and parse the source base of the
target application. The output of this step is an in-memory representation of the program in the form of a set of abstract syntax trees (ASTs), one for each application source file. Users can then visualize these ASTs to better understand their structure, or the ASTs can be exposed to external software engineering tools via an API interface. These software engineering tools use the API to perform the specific language processing tasks desired for that tool. If a software engineering tool uses the API for instrumentation, the modified ASTs corresponding to the instrumented source are then provided as input to the source code regeneration visitor, which transforms each of the modified ASTs (back) into nesC source files – files that are ready to be compiled.

4.1.1 Implementation Technologies

Third party tools were used to automate the generation of the scanner and parser. These tools include *JFlex* [60], a Java-based scanner generator, and *CUP* [55], a Java-based parser generator. The generated scanner recognizes approximately 140 different character sequences corresponding to 108 different tokens in the nesC language. These tokens were identified through a careful examination of the scanner used in *nescc* [7], the nesC compiler. The tokens are passed to the generated parser, which consists of 108 terminals, 214 non-terminals, and 588 productions. The terminals, non-terminals, and productions were modeled after the Bison grammar for nesC and expanded to support source code regenera-
tion. Specifically, the grammar was modified to support the `#include` preprocessor directive. These directives are normally eliminated by the preprocessor prior to parse time; however, preprocessing nesC source files that are to be regenerated results in programs that contain duplicated symbols at compile time (e.g., when multiple source files include the same header file). To address this problem, we extract and apply all `#define` macros, integrate the symbols defined in the included file into the symbol table, and retain the `#include` statements in the ASTs.

To simplify code traversal and modification, the parser reduces syntactic variation by performing several AST-level transformations while reading each source file. The goal of these transformations is to normalize programs – to take a set of syntactically different, yet functionally equivalent, source expressions and convert them into a single, easy-to-use syntactic form. The following paragraphs describe each of these normalization techniques and discuss the benefits that they provide.

**Compound Statement Normalization.** The toolkit adds explicit compound statements for all control structures. Introducing the compound statements simplifies the task of inserting code segments into a program. Without the transformation, when developers want to insert a line of code within a control structure, they must first ensure that the body of the structure is a compound statement, and if not, create one.

**Return Statement Normalization.** The toolkit inserts an explicit return statement at the end of all `void`-returning functions. Introducing the return statement simplifies the task of identifying (and navigating to) the exit points of a function. (Other exit points may already exist as explicit return statements within the function.) Without the transformation, when developers want to identify all exit points from a function, they must first identify all return statements and if the function is a `void` returning function, they must identify the last statement executed within the function.

**Wiring Statement Normalization.** The toolkit transforms all “right-to-left” wirings (e.g., B.I ← A.I) to “left-to-right” wirings (e.g., A.I → B.I) within configurations. Introduc-

---

1Identifying the last statement executed with a function is non-trivial. Consider the case where a switch statement is found at the end of a non-void function. Each case of the statement would need to be considered.
ing this transformation simplifies the task of identifying the users and providers of interfaces. Without the transformation, when developers want to identify the set of components that use or provide a given command, they must consider the direction of the wiring operator and the meaning of the direction in terms of the components involved. (Recall that the wiring operator always “points” from the user of an interface to the provider.)

**Interface Name Normalization.** The toolkit resolves the fully-qualified interface names of all components used within configurations. For example, a wiring $A \rightarrow B.I$ would be transformed to $A.I \rightarrow B.I$. Introducing the fully-qualified interface names simplifies the task of identifying the set of components that are wired to a given user. Without the transformation, when developers need to know which component realizes a used interface, they must resolve all of the wirings and search for a matching uses-provides pair.

**Component Reference Normalization.** The toolkit renames all component references that are not explicitly renamed. For example, a component reference $MainC$ is transformed into $MainC$ as $MainC$. Introducing this transformation simplifies the task of identifying the component associated with a given name. Local names can be mapped to their corresponding actual component names. Without this transformation, users must include special checks in their code to determine whether a component name represents an actual component name, or a local renaming.

**List Normalization.** Finally, the toolkit inserts *dummy* elements into empty lists within ASTs. These lists correspond to many of the syntactic elements that can occur 0 or more times within the program source (e.g., uses statements). Consider, for example, a AST node corresponding to a top-level configuration (i.e., a configuration that neither uses nor provides interfaces). Prior to this transformation, the node’s reference to its used interface lists would be null. After the transformation, however, the reference “points” to a dummy list node. Introducing these dummy elements simplifies the task of navigating to and inserting new elements into the lists. Without the transformation, developers would be required to include special cases in their code to check for empty lists and to create them as necessary.
<table>
<thead>
<tr>
<th>Return Type</th>
<th>Method Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConfigComp</td>
<td>findConfigUsing(String compName, Map sourceFiles)</td>
</tr>
<tr>
<td>List</td>
<td>findAllConfigsUsing(String compName, Map sourceFiles)</td>
</tr>
<tr>
<td>AstNode</td>
<td>getParent()</td>
</tr>
<tr>
<td>List</td>
<td>getNodesOfType(Class type, boolean recursive)</td>
</tr>
</tbody>
</table>

Table 4.1: Traversal Methods (partial)

Each of these AST-level transformations supports our design goal of enabling developers to create new software engineering tools easily. Without these normalizations, analysis and instrumentation tasks are still possible, however users would be required to contend with more syntactically diverse ASTs, requiring more special-case checks throughout their code.

4.1.2 API Details

The most fundamental feature provided by the toolkit is the ability to parse a nesC source base. While it is possible to process individual files, many applications benefit from a configuration-based parse: The toolkit provides a method to initiate a parse from a specified configuration. The method accepts the configuration path as argument and a list of search paths used to locate dependent components identified during the parse. The result is a set of ASTs corresponding to the source files processed during the parse. These trees can be processed manually using standard accessor methods. Alternatively, the toolkit provides a set of convenience methods to simplify the most common tasks, again supporting the goal of enabling users to easily develop new software engineering tools. Tables 4.1, 4.2, and 4.3 include representative lists of these methods.

First, Table 4.1 lists four representative traversal methods. These methods simplify static analysis tasks by enabling developers to navigate through the ASTs to collect information about the structure of the programs (e.g., what modules exist in the system, which functions does each module contain, which other functions does each function call). They

---

2Conceptually the result is a set; however, to simplify lookup tasks the method returns a mapping from file name (component name) to AST.
3Some of the type and function names have been shortened for presentation.
also simplify instrumentation tasks by enabling developers to navigate to the location in the ASTs where the instrumentation will be performed.

`findConfigUsing()` takes a component name and a mapping of file names to AST nodes and returns an AST node corresponding to the first configuration component in the mapping that imports the specified component name (i.e., the first AST corresponding to a configuration component found in the mapping that imports the component with the specified name). This method is useful, for example, when a new uses interface is added to a component and it is necessary to wire a realizing component to that interface. Similarly, `findAllConfigsUsing()` returns a list of all AST nodes corresponding to configurations that import the specified component name. This method is useful, for example, when a user wants to remove an interface from a component and it is necessary to identify all configurations that may have wired a component to that interface. Calls to these methods require a component name (e.g., "BlinkC") and the mapping generated by the parsing process. The `getParent()` method has the obvious meaning and is useful in discovering the containing context associated with an AST node.

The final traversal method in the table, `getNodesOfType()`, is especially interesting. The method searches for AST nodes of a specified `class` type. This approach simplifies the process of identifying sub-trees within an AST. The method returns a list of AST nodes matching the specified class type rooted at the invocation target. If the desired search is recursive, a full traversal of the tree is performed; otherwise, only the direct children of the invocation target are included in the search. This method is useful, for example, in identifying the modules referenced by a configuration. This is achieved by passing a `Class` corresponding to the type for which to search (e.g., `Component.class`) and a boolean indicating whether the search should be performed recursively. Each of these methods enables users to traverse the in-memory representations of nesC programs easily.

It is interesting to note that Java’s reflection capabilities simplify the implementation of the `getNodesOfType()` traversal method. Pseudocode for the method is shown in Listing 4.1. The method is defined in `AbstractAstNode`, the abstract base class for all
AST nodes, enabling users to apply the method to any AST within a system. First, the method uses Java’s generic mechanism to place a constraint on the caller, ensuring that callers search only for class types that extend AstNode (Line 2). Next, the method identifies a list of all of the non-static fields defined in the activated object’s class type (Lines 6–22). For each non-null field, the method tests to determine if the field is of the desired type (Line 9). If the object is of the desired type, the method adds the object to the result list (Line 10). If the recursive flag is set, the method is called recursively on the associated field (Lines 11–13). If the field is not of the desired type, a check is made to determine whether the object is an array or a Collection. If the object is an array or a collection, the method iterates over each element (Lines 15–21). Again, if the type of the element matches the desired type, the method adds the sub-element to a result list (Line 17) and if the recursive flag is set, the method is called recursively on the object (Lines 18–21).

Table 4.2 lists five representative modification methods. addComponentToConf() is used to introduce a new component reference in an existing configuration. The method
### Table 4.2: Modification Methods (partial)

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Method Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>addComponentToConf(ConfigComp conf, String name, String as)</td>
</tr>
<tr>
<td>void</td>
<td>addUses(Component component, Uses uses)</td>
</tr>
<tr>
<td>void</td>
<td>addProvides(Component component, Provides provides)</td>
</tr>
<tr>
<td>void</td>
<td>addWiring(ConfigComp conf, Connection connection)</td>
</tr>
<tr>
<td>Map</td>
<td>instantiateGenerics(Map originalASTs, String topLevelFname)</td>
</tr>
</tbody>
</table>

takes the configuration component to which to add the new reference, the actual name of the component, and a string used to rename the component locally. The method is achieved by first acquiring a reference to a configuration component AST node, by invoking `findConfigUsing()` for instance, and then passing to the method that reference, the name of the component, and the desired local name. `addUses()` is used to introduce a new used interface within a component. The method takes a target component and an AST corresponding to a nesC `uses` statement. `addProvides()` is similarly used to introduce a new provided interface within a component. Calls to `addUses()` and `addProvides()` are achieved by passing a reference to a `Component` (e.g., by calling `findConfigUsing()`\(^4\)) and a reference to a `Uses` or `Provides`. (We shall show how the API enables users to generate `Uses` and `Provides` objects later in this section.) `addWiring()` is used to add a wiring statement to an existing configuration. The method accepts the configuration component to which to add the wiring and an AST corresponding to a nesC wiring statement. These methods are useful, for example, to inject new services into existing applications. `addProvides()` is used to add a new provided interface to a component. Similarly, `addUses()` is used to enable a component to use the services provided by another component. `addComponentToConf()` is used to import additional components into a configuration. After components have been added to a configuration using `addComponentToConf()`, `addWiring()` is useful for adding wirings between the used and provided interfaces to the newly introduced components.

The final method in the table, `instantiateGenerics()`, is among the most complex operations provided by the toolkit. The method transforms a set of ASTs to eliminate generic modules and generic configurations. The approach is to duplicate the generic types  

\(^4\)ConfigComp is a subclass of Component.
by substituting actual arguments for formal generic parameters. The instantiated components are automatically renamed to include a unique integer tag to prevent name collisions. All component references within configurations are updated to reflect the new names. This process is analogous to transforming a C++ source base containing template classes into an equivalent source base without templates. This method is useful because it provides tool developers an analysis and instrumentation model that mirrors the semantics of the underlying programming model. It enables developers to analyze and modify the products of the instantiation process. Each of the modification methods directly supports our goal of simplifying the programmatic modification of nesC programs by providing a familiar instrumentation model, and providing convenience methods that simplify the most common instrumentation tasks.

Finally, Table 4.3 lists eight representative generation methods. These methods are similar to the methods provided by the CodeDOM [75] API included with C#. `generateExprStmt()` accepts a string containing a nesC expression statement and returns an AST corresponding to that statement. This method is useful, for example, for generating an AST corresponding to a function call. The newly created sub-tree can then be added to an existing AST. `generateEnum()` accepts a string containing a C-style enumeration declaration and returns an AST corresponding to that declaration. `generateIfStmt()` accepts a string containing an integral expression and returns an AST corresponding to an empty if statement with that expression as its condition. `generateVarDec()` accepts a string containing the text corresponding to a variable declaration and returns an AST corresponding to
private static ExprStmt generateExprStmt(String expressionString) {
    ByteArrayOutputStream os = new ByteArrayOutputStream();
    PrintWriter out = new PrintWriter(os);
    out.printf("void foo() { %s; }", expressionString);
    out.close();
    Lexer lexer = new Lexer(new ByteArrayInputStream(os.toByteArray()));
    Parser parser = new Parser(lexer, null);
    parser.parse();
    List<Expr> expressions = parser.getRootDispatch().getNodesOfType(Expr.class, true);
    return new ExprStmt(expressions.get(0));
}

Listing 4.2: Sample Generation Method

that declaration. The generateFunctionDef() method accepts a string corresponding to a
function signature and returns an AST corresponding to an empty function body with that
signature. generateComponentRef() accepts a string corresponding to a component refer-
ence within a configuration and returns an AST corresponding to that component reference.
The generateWiring() method accepts a string corresponding to a nesC wiring statement and
returns an AST corresponding to the wiring statement. Finally, the generateModule() ac-
cepts a module name and returns an AST corresponding to an empty module with that
name. Rather than simply modifying an existing AST, this method can be used to create
new modules programmatically. The generation methods make it easier for users to gener-
ate ASTs that can be inserted into an existing source base programmatically, simplifying
instrumentation tasks. Without these methods, developers would need to be much more
familiar with the structure of the ASTs they want to create. Each object would need to be
created, building the AST fragment from the bottom up towards the top.

As mentioned previously, the toolkit includes only methods that simplify the most
common tasks. Users may need to develop custom methods to meet their needs. Listing 4.2
illustrates the general form of a representative generation method – specifically, the source
code for generateExprStmt(). The method begins by creating an in-memory output stream
and wrapping the stream with an object that exposes methods to print strings (Lines 2–3).
Next, the method creates a valid top-level nesC parse target where the desired code segment,
here an expression, can exist (Line 5). In the listing, the parse target takes the form of a C header file\(^5\). The function includes the string form of the code segment within its body. Next, the contents of the underlying output stream are sent to an instance of the nesC lexer and parser and then parsed (Lines 8–10). The method then uses `getNodesOfType()` to access a list\(^6\) of objects corresponding to the code segment of interest (Lines 12–13). Finally, the method returns the AST corresponding to the code segment of interest (Line 15). The caller of this method can then insert the returned object into an existing AST as part of some instrumentation activity.

The implementation of this method might seem rather simple – and it is. The reason for this is that the complexity of building the AST for the code segment is hidden within the API. Additionally, `getNodesOfType()` provides an easy way to navigate to the appropriate sub-tree within the AST produced by the parser. Whether using existing methods or developing custom methods, nAIT’s design makes it easier for users to traverse, modify, and generate the in-memory representations of nesC programs.

### 4.1.3 AST Explorer

The final component of the toolkit implementation is a Java-based visualization tool for exploring ASTs. The **AST Explorer** was developed using [JUNG][84], a software library for visualizing graphs. The tool takes as input any AST node and displays a graphical representation of the AST rooted at that node. The tool uses nAIT’s traversal methods to walk the AST and generate the visualization.

Figure 4.2 includes a sample screen capture of the AST Explorer. The figure shows a portion of an AST corresponding to a module state variable declaration. Internal nodes are labeled using the class name of the corresponding AST node. For example, the class type of the root node in the figure is `JustDatadef`. The label associated with generic AST nodes, such as `Pair` in the figure, includes the names of all type parameters, or `null` if a parameter is null. Leaf nodes are labeled using the text that appears in the target source file. In the

---

\(^5\)nesC interfaces, modules, and configurations are also valid parse targets.

\(^6\)In this case, a list containing only one element.
Figure 4.2: AST Explorer Screen Capture

Figure, uint8_t and data are leaf nodes and correspond to the type and name of the variable in the source, respectively. Edges in the tree are labeled using the corresponding variable name of the containing AST node. For example, in the figure, the JustDatadef class contains two fields, typeElementList and declaration.

The buttons at the bottom of the figure enable users to control the behavior of the visualization. The “collapse” button enables users to filter out unwanted information by collapsing a subtree rooted at a selected node into a single node. The “expand” button enables users to view the subtree rooted at a previously collapsed node. The “toggle mode” button is used to switch between translation and selection modes. The translation mode enables users to move the graph as a whole within the window. The selection mode enables users to select and move individual nodes within the graph. These modes enable users to move and modify the display, making the generated tree easier to comprehend.
The AST Explorer enables developers to understand the structure of the ASTs more easily and is used throughout the development process. When a user is interested in traversing an AST to identify, for example, a variable declaration within a function, she uses the AST Explorer to visualize a similar code segment. From the root of the visualization, down to the variable declaration, the user can easily understand the AST’s node containment hierarchy. This information helps the user to determine which AST traversal technique is most appropriate. When a user is interested in modifying an AST, again she uses the AST Explorer to visualize a similar code segment. Understanding the structure of the AST enables the user to identify the set of classes that are involved in the desired changes. Finally, when a user develops her own generation method, she uses the AST Explorer to determine which portion of the generated AST contains the subtree of interest. An example of how the tool aids in AST comprehension will be seen when we describe representative use-cases in Section 4.2.

### 4.2 Use-Case Scenario

Next, we turn our attention to a use-case scenario that illustrates the features and benefits of nAIT’s design and outlines how the toolkit’s traversal, modification, and generation methods can be used to develop new source-based software engineering tools for nesC. The use-case describes the development of a source-based state predicate evaluation tool.

The *Predicate Evaluator* is a tool that enables developers to monitor up to 3 state predicates defined in terms of the state elements of a nesC program to detect invariant violations. Each predicate is associated with the state of an LED on the hosting device; *off* indicates that the predicate has *always* evaluated to true, *on* indicates that the predicate has, *at least once*, evaluated to false. The tool presents users with a graphical interface that enables them to select the top-level configuration of the target application, select the variables associated with each LED, and define the associated predicates in terms of those variables. Once the predicates have been defined, the tool instruments the nesC source
base to monitor the selected variables for changes. When a nesC function can potentially modify one or more of the selected variables, the function is instrumented to reevaluate the associated predicates prior to returning to the calling function. If a predicate evaluates to false, the associated LED is activated to indicate the error condition. The remainder of this subsection outlines the process of applying the tool to a nesC source base and describes how nAIT’s traversal, modification, and generation methods are used to simplify the tool’s implementation.

4.2.1 Tool Overview

The process begins with the user starting the graphical application. A screen capture of the application window is shown in Figure 4.3. Next, the user selects the top-level configuration of the target application using the “load” button located at the bottom left of the window. After the top-level configuration is selected, the path to the associated source file is shown in the text field along the bottom of the window, and the tool uses nAIT to
parse the application source materials. The tool then uses nAIT to search the resulting set of ASTs for all modules defined within the system, all state variables defined within each module, and the type information for each of the state variables. Once the tool collects this information, it updates the “program symbols” list on the left side of the window to display a list of all state variables within the system. Each element in the list includes the type of the associated variable, followed by a mangled variable name that is used to fully-qualify the context in which the name appears. This mangled name begins with an asterisk, indicating that a pointer to the variable will be available when defining the predicate. If the variable itself is a pointer, additional asterisks will follow. Next, the mangled name includes the name of the containing module, followed by an underscore and the variable name. Elements that represent arrays are additionally followed by brackets. For example, the first element visible in the figure is a variable of type uint8_t named available defined in module HeapP.

Once the tool parses the program source and populates the program symbols list, the user selects an LED from the set of radio buttons in the top left section of the window. (Recall that each LED corresponds to a predicate.) Once the user selects an LED, she selects one or more of the variables in the “program symbols” list. A pointer to each selected variable is available when defining the associated predicate. In the figure, HeapP_available, HeapP_capacity, and TestP_used are selected.

Next, the user defines the predicate as a boolean expression defined in terms of the selected variables (in the text area in the right side of the window). The predicate in the figure is used to ensure that the number of memory segments available in a heap, plus the number of segments used by an application equals the total number of segments available. If, at runtime, the predicate does not hold true, one of three error conditions exists: Either a memory segment is lost (i.e., a memory leak exists), a memory segment has been deallocated multiple times, or a bookkeeping error exists in the target application.

7The “program symbols” list excludes volatile variables whose values can be changed directly by the hardware.

8In this example, TestP is the only component using the heap. If other components were also using the heap, their heap usage would also be available as an argument to this predicate.
Once a predicate is defined, the user can define additional predicates in a similar way using the remaining two LEDs. Once the user defines all the predicates, she uses the “save” button in the bottom right hand corner of the window to start the instrumentation process. The tool then uses nAIT to instrument the modules containing the selected variables and to regenerate the nesC source. The regenerated source can then be compiled and installed on a device for testing.

4.2.2 Tool Implementation

The Predicate Evaluator’s implementation is based on the services provided by nAIT, and was guided by nAIT’s AST Explorer. The following paragraphs outline the implementation of the tool and explain how the AST Explorer was used to guide specific portions of the implementation.

When the user selects a top-level configuration from the graphical interface, the Predicate Evaluator uses nAIT to perform a configuration-based parse of the nesC source materials rooted at the selected file. This parsing process returns a set of ASTs, each corresponding to a source file. Next, the Predicate Evaluator searches this set for ASTs that correspond to nesC modules. For each AST, the tool uses the traversal method findNodesOfType() to determine if the current AST corresponds to a module. (The method returns an empty list when no AST nodes of the specified type are found.) For each AST containing a module, getNodesOfType() is again used to identify all of the variable declarations within the module. For each variable, traditional accessor methods associated with the AST nodes are used to collect the type information, determine whether the variable is a pointer, and determine whether the variable is an array. The method that realizes this behavior is only 49 lines of code. nAIT enables tool developers to collect information about target programs quickly and easily.

During the creation of the Predicate Evaluator, we used the AST Explorer to understand how the type, pointer, and array information is represented in the ASTs corresponding
Figure 4.4: AST Explorer – State Variable Declaration

to state variable declarations. Figure 4.4\(^9\) shows a screen capture of an AST Explorer window that includes the sub-tree corresponding to the declaration of a module state variable of type uint8_t, named data. The node at the root of the subtree, JustDatadef corresponds to the AST node that represents a data definition. A data definition is comprised of a list of elements representing the type of the data, typeElementList, and a declaration. Type element lists include information related to a variable’s type and storage class. For example, static unsigned long int produces an element list of length 4, one element for each token. The type element list in the figure contains only one entry, a TypenameTypeElement. Typename type elements represent types that are introduced using the typedef keyword. The token associated with the TypenameTypeElement is uint8_t, the type of the declared variable. The declaration chain associated with the root JustDatadef is similar. It begins with an Initdecls, a declaration that can include a declaration-time initializer. The declaration consists of a

\(^9\)Figure 4.4 is a reproduction of Figure 4.2, included here for the reader’s convenience.
generic Pair, associating a list of gcc attributes with each declaration. In the figure, no gcc attributes are associated with the declared variable, therefore the attribute list is empty. The second part of the Pair, Initdcl, corresponds to a single declaration (again, which can include an initializer). The type of the declaration is an IdentifierDeclarator, indicating that the declaration is a variable declaration. Finally, the name of the declared variable is data. Had the variable been a pointer or an array, additional nodes would have existed in the associated AST to represent that information. In the case of a simple variable declaration illustrated in the figure, the AST consists of 9 AST nodes; more complex expressions consist of dozens of nodes. The AST Explorer enables developers to comprehend the structure of ASTs and provides insights into the structure of the ASTs that would otherwise be difficult to glean without visualization support.

After the user selects “save”, the tool uses nAIT’s generation methods to create three new nesC source files which encapsulate the predicate evaluation logic. The first, PredicateEvaluator.nc, contains a nesC interface to the predicate evaluation component. The tool uses nAIT to generate the interface and to add all necessary command prototypes to the interface. The process of adding these prototypes involves three lines of code, each invoking a generation method and using local AST methods to add the generated code to the interface. The interface exposes commands used to register the address of each user-selected state variable with the component, as well as commands used to reevaluate each of the three predicates. The second generated source file is PredicateEvaluatorM.nc. This source file, generated using the generateModule() API method, contains a nesC module that provides the previously described PredicateEvaluator interface. The module includes state variables that are used to store pointers to each of the user-selected variables, as well as the implementations of the commands defined in the PredicateEvaluator interface. The generateVarDec() and generateFunctionDef() API methods are used to generate the variables and commands, respectively. The process of creating these variables and commands involves a simple loop that iterates over the selected variables, a call to a generation method to create the new AST sub-tree, and a call to a local AST modification method to incorpo-
rate the sub-tree. The Predicate Evaluator uses the user-defined predicates to implement
the bodies of the commands associated with each predicate. If the predicate evaluates to
false, the implementation activates the corresponding LED. The final generated source file
is `PredicateEvaluatorC.nc`. This source file contains a nesC configuration that re-exposes
the `PredicateEvaluator` interface provided by `PredicateEvaluatorM` and wires the standard
implementation of the `Leds` interface to `PredicateEvaluatorM`. The implementation of the
method to generate this file consists of only 12 lines of code. Finally, the source of each
of the three newly generated files is generated using the nesC source regeneration visitor,
included with nAIT.

Once the source files for the newly generated interface, module, and component are
generated, the existing application source base must be instrumented to include calls to
register the user-selected state variables with the newly generated component. First, each
module containing a selected variable is modified to use the generated `PredicateEvaluator`
interface. Using this interface enables each module to call the commands provided by the
interface. A generation method is used to create an appropriate AST corresponding to the
uses statement, and the modification method `addUses()` is used to add the newly generated
statement to the uses list of the module. Similarly, the module is modified to use the `Boot`
interface, appropriately renamed to avoid conflicts. Next, the `generateFunctionDef()`
generation method is used to create an implementation of the `booted()` event associated
with the `Boot` interface. Recall that the `booted()` event is signaled when the underlying
hardware platform has been initialized and the system is ready for execution. The body
of this function is populated with calls to register each of the user-selected variables with
the `PredicateEvaluator` component. These calls are generated using the `generateExprStmt()`
API method. The generated `booted()` event is then added to the module using AST-level
modification methods. The toolkit method responsible for populating the body of the
`booted()` method with registration calls consists of only 18 lines of code.

After the code base is instrumented to register the variables of interest with the `PredicateEvaluator` component, the functions that modify the variables must be instrumented
to include calls to reevaluate the predicates. First, `findNodesOfType()` is used to identify all functions within the containing module’s AST. Next, the `findNodesOfType()` method is again used to search within each function AST for nodes corresponding to assignment statements and pre- and post-increment and decrement operations that change the state of the user-selected variables\(^{10}\). If a function is found to modify any of the selected variables, the `getNodesOfType()` traversal method is used to find all ASTs representing return statements. (Recall that nAIT-normalizes the code by automatically including explicit returns at parse time.) The `getParent()` traversal method is then used to identify the parent of the corresponding return node. The `generateExprStmt()` method is used to generate a call to the appropriate reevaluation command exposed by the `PredicateEvaluator` interface, and AST-level methods are used to insert the generated function calls into the parent node, immediately prior to the return statement. The code required to instrument the target source with calls to reevaluate the predicates required fewer than 200 lines of code.

Once the modules have been fully instrumented, the associated nesC configurations must be updated to wire the newly introduced `PredicateEvaluator` interface to the `PredicateEvaluatorC` component. The `findConfigUsing()` API method is used to identify a configuration that uses the modified module. The `addComponentToConf()` API method is used to add `MainC` (renamed to avoid conflicts) and `PredicateEvaluatorC` to the configuration’s component list. `MainC` is used to signal the newly introduced `Boot.booted()` event on startup, and `PredicateEvaluatorC` is used to provide the implementation of the introduced `PredicateEvaluator` interface. The `generateWiring()` API method is used to generate the appropriate wirings from the module to the components, and `addWiring()` is used to update the ASTs. This process required less than 30 lines of code. The source regeneration visitor is then used to regenerate the source of all modified modules.

nAIT simplified the development of the Predicate Evaluator. The services provided by the toolkit enable the Predicate Evaluator to traverse the ASTs corresponding to nesC source materials and to analyze the trees to identify the modules and state variables within

\(^{10}\)Note that this approach only supports directly modified variables; variables modified via indirection are not supported.
the system. The toolkit also enables the Predicate Evaluator to generate new and instrument existing nesC source materials easily. The implementation of the Predicate Evaluator consists of approximately 1500 lines, including comments and blank lines. The majority of this code is associated with the creation and management of the graphical user interface. The complexity associated with the nesC language is hidden by nAIT, and the implementation of the Predicate Evaluator focuses on the software engineering tasks of interest.

4.3 Evaluation

nAIT is a toolkit that enables developers to perform static analysis and instrumentation activities on embedded network systems. In this section, we evaluate the toolkit in terms of the execution time required to load, traverse, and modify the ASTs, as well as the associated memory overhead. Our evaluation of the toolkit is based on the following objectives. The toolkit must:

- **O2.1** Enable hardware platform-independent software engineering tools that target the nesC platform to be developed quickly and easily
- **O2.2** Be suitable for the development of both interactive and batch-based software engineering tools
- **O2.3** Enable analysis and instrumentation tasks to be performed on desktop-class hardware

All experiments were conducted on an Intel Pentium 4 processor running at 2.8 GHz with hyperthreading technology and 2 GB of main memory. The hosting operating system was GNU/Linux 2.6.23 (Gentoo) with simultaneous multithreading enabled and version 2.6.1 of the GNU C standard library. The hosting Java virtual machine was Sun’s Standard Edition Runtime Environment, version 1.6.0.04-b12. All applications targeted the Telosb mote platform using TinyOS-2.x from CVS, downloaded on August 1, 2007.
The following subsections detail our evaluation of nAIT. Subsection 4.3.1 details the properties of nAIT that make it suitable for the development of software engineering tools for the nesC platform. Subsection 4.3.2 details the time required to parse, traverse, and instantiate a set of standard nesC applications. Finally, Subsection 4.3.3 details the space requirements to represent the same programs in RAM.

4.3.1 Suitability of Purpose

This section highlights the properties of nAIT that enable it to provide developers with a platform-neutral analysis and instrumentation framework that is quick easy to use. First, the toolkit is source-based. The source-based approach enables the toolkit to be used in analysis and instrumentation tasks for any mote (or simulator) platform capable of running nesC applications. This makes new software engineering tools developed using the toolkit immediately applicable to all nesC-based embedded network systems. The toolkit provides an API that simplifies the development of such software engineering tools. The traversal methods enable users to navigate to “interesting” portions of the underlying ASTs quickly and easily, with little code. Once such a portion has been identified, the modification methods enable users to make changes to the existing ASTs. Users can also use the generation methods to create new program segments to be inserted into the ASTs. Without such methods, traversal, modification and generation activities would be more difficult and time-consuming. Traversal tasks would require users to directly follow parent-child links within ASTs, requiring users to be much more familiar with the AST structure. Similarly, modification and generation tasks would involve more error-prone, low-level AST manipulation. To further simplify the traversal, modification, and generation tasks, nAIT includes an AST browser that enables users to visualize the structure of ASTs. This browser allows users to identify portions of ASTs to which they wish to traverse and provide them with context information that is necessary to modify or generate sub-trees. These properties make the toolkit suitable for the quick and easy development of hardware platform-independent software engineering tools – the purpose for which it was intended (O2.1).
4.3.2 Time Requirements

The following paragraphs detail our analysis of the time required to perform various tasks with nAIT. Specifically, we consider the time required to (1) parse programs, (2) use traversal methods to gather static information about programs, and (3) instantiate generic components within programs. The objective is for the toolkit to be suitable for the development of both interactive and batch-based software engineering tools.

Parse Time. The parse time of nAIT plays an important role in assessing the toolkit’s utility. If the parse time is high, the toolkit will not be suitable for tasks involving frequent or repetitive execution. To evaluate the parse time, we developed a test application to measure the time required to parse each source file in a target application. Dependencies among source files cause the parse of a source file to be suspended while dependent source files are parsed. This results in the recorded times for each file including not only the time for that file, but all its unparsed dependents\(^{11}\). To calculate the per-file parse times, the parse times were recursively calculated and adjusted to remove the time required for dependent files. Also note that values include the time required to scan the filesystem for the source files, to run the C preprocessor on those files, and to perform the automated source transformations described in Section 4.1.

A summary of the experimental results is shown in Table 4.4. Each row corresponds to one of the programs included with the standard TinyOS distribution. The Files column corresponds to the number of files scanned, including files that contain only preprocessor macros (not represented in ASTs). The remaining columns have the obvious meanings. Despite the significant variation in parsing time witnessed during each experiment, the results are favorable. In the worst case, the toolkit requires approximately 22 seconds to scan and parse the largest project, MultihopOscilloscopeLqi. This makes the toolkit well-suited for the development of both interactive and batch-based software engineering tools (O2.2).

\(^{11}\)Each file is parsed once. If a file depends on another file that has already been parsed, the dependent file is not re-parsed.
<table>
<thead>
<tr>
<th>Application</th>
<th>Files</th>
<th>Min</th>
<th>Max</th>
<th>Median</th>
<th>Mean</th>
<th>Stddev</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseStation</td>
<td>197</td>
<td>0.00032</td>
<td>1.77748</td>
<td>0.00983</td>
<td>0.07335</td>
<td>0.20582</td>
<td>14.44890</td>
</tr>
<tr>
<td>Blink</td>
<td>71</td>
<td>0.00039</td>
<td>0.52721</td>
<td>0.01393</td>
<td>0.05963</td>
<td>0.10607</td>
<td>4.23374</td>
</tr>
<tr>
<td>MViz</td>
<td>252</td>
<td>0.00032</td>
<td>2.17438</td>
<td>0.01094</td>
<td>0.07417</td>
<td>0.23998</td>
<td>18.69143</td>
</tr>
<tr>
<td>MultihopOscilloscope</td>
<td>278</td>
<td>0.00031</td>
<td>2.37474</td>
<td>0.01152</td>
<td>0.07391</td>
<td>0.22812</td>
<td>20.54766</td>
</tr>
<tr>
<td>MultihopOscilloscopeLqi</td>
<td>267</td>
<td>0.00029</td>
<td>2.25911</td>
<td>0.01000</td>
<td>0.08113</td>
<td>0.23519</td>
<td>21.66255</td>
</tr>
<tr>
<td>Null</td>
<td>41</td>
<td>0.00039</td>
<td>0.49458</td>
<td>0.00533</td>
<td>0.06797</td>
<td>0.11359</td>
<td>2.78665</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>211</td>
<td>0.00030</td>
<td>1.89563</td>
<td>0.01267</td>
<td>0.07428</td>
<td>0.21645</td>
<td>15.67282</td>
</tr>
<tr>
<td>Powerup</td>
<td>51</td>
<td>0.00039</td>
<td>0.50285</td>
<td>0.01467</td>
<td>0.06393</td>
<td>0.10936</td>
<td>3.26032</td>
</tr>
<tr>
<td>RadioCountToLeds</td>
<td>183</td>
<td>0.00029</td>
<td>1.60355</td>
<td>0.01190</td>
<td>0.08567</td>
<td>0.23989</td>
<td>15.67829</td>
</tr>
<tr>
<td>RadioSenseToLeds</td>
<td>211</td>
<td>0.00028</td>
<td>1.88268</td>
<td>0.01166</td>
<td>0.07855</td>
<td>0.20683</td>
<td>16.57465</td>
</tr>
<tr>
<td>Sense</td>
<td>107</td>
<td>0.00039</td>
<td>0.84337</td>
<td>0.01285</td>
<td>0.06706</td>
<td>0.13333</td>
<td>7.17532</td>
</tr>
</tbody>
</table>

Table 4.4: Source File Parsing Times

**Traversal Time.** The traversal time also plays an important factor in assessing the toolkit’s utility. If the time required to traverse ASTs is high, the toolkit will again not be suitable for tasks involving frequent or repetitive execution. To evaluate the traversal time, we developed a test application to measure the time required to invoke the getNodes-OfType() method and to report information about the structure of those programs. This information includes the names of all modules within the system, the names of all functions within each module, and the names of all functions called by each function in each module.

A summary of the experimental results is shown in Table 4.5 and Figure 4.5. In the table, values within the Modules column corresponds to the time required to identify only the names of the modules within the system. The Functions column corresponds to the time required to identify the names of the modules within the system, as well as the names of all the functions defined within those modules. Finally, the Calls column corresponds to the time required to identify the names of the modules, the names of the functions within those modules, and the names of all the functions called by each of the functions. As can be seen in the figure, the time required to identify the module names is insignificant. This is because the corresponding information is stored at the root of the AST – a deep traversal is unnecessary. Identifying the functions within each module and the calls within each function is more time consuming. As shown in the table, MultihopOscilloscope, at approximately 3 seconds, required the most time to collect and print the information.
(Recall from Table 4.4 that MultihopOscilloscope is the program with the largest number of files.) As we shall see later in this section, modules typically produce the largest ASTs. Therefore, this result indicates that the toolkit is well-suited for the development of both interactive and batch-based software engineering tools (O2.2).

**Instantiation Time.** The *instantiation time* also plays a role in assessing the toolkit’s utility. If the time required to instantiate generic components is high, the toolkit will be unsuitable for tasks involving frequent or repetitive execution. We evaluate instantiation time by invoking the `instantiateGenerics()` modification method. The results include the

<table>
<thead>
<tr>
<th>Application</th>
<th>Modules</th>
<th>Functions</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseStation</td>
<td>0.00301</td>
<td>1.01937</td>
<td>2.13567</td>
</tr>
<tr>
<td>Blink</td>
<td>0.00145</td>
<td>0.23598</td>
<td>0.41837</td>
</tr>
<tr>
<td>MViz</td>
<td>0.00364</td>
<td>1.39871</td>
<td>2.58866</td>
</tr>
<tr>
<td>MultihopOscilloscope</td>
<td>0.00413</td>
<td>1.63326</td>
<td>3.05228</td>
</tr>
<tr>
<td>MultihopOscilloscopeLqi</td>
<td>0.00410</td>
<td>1.46101</td>
<td>2.76254</td>
</tr>
<tr>
<td>Null</td>
<td>0.00098</td>
<td>0.12717</td>
<td>0.22484</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>0.00375</td>
<td>1.06499</td>
<td>2.03419</td>
</tr>
<tr>
<td>Powerup</td>
<td>0.00115</td>
<td>0.16078</td>
<td>0.28243</td>
</tr>
<tr>
<td>RadioCountToLeds</td>
<td>0.00278</td>
<td>0.81639</td>
<td>1.58281</td>
</tr>
<tr>
<td>RadioSenseToLeds</td>
<td>0.00355</td>
<td>1.05274</td>
<td>1.98813</td>
</tr>
<tr>
<td>Sense</td>
<td>0.00194</td>
<td>0.46561</td>
<td>0.97985</td>
</tr>
</tbody>
</table>

Table 4.5: AST Traversal Times
Table 4.6: Generic Instantiation Times

<table>
<thead>
<tr>
<th>Application</th>
<th>File Count</th>
<th>Inst. Count</th>
<th>New Files</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseStation</td>
<td>197</td>
<td>257</td>
<td>60</td>
<td>10.93</td>
</tr>
<tr>
<td>Blink</td>
<td>71</td>
<td>121</td>
<td>50</td>
<td>7.38</td>
</tr>
<tr>
<td>MViz</td>
<td>252</td>
<td>320</td>
<td>68</td>
<td>15.90</td>
</tr>
<tr>
<td>MultihopOscilloscope</td>
<td>278</td>
<td>359</td>
<td>81</td>
<td>18.37</td>
</tr>
<tr>
<td>MultihopOscilloscopeLqi</td>
<td>267</td>
<td>345</td>
<td>78</td>
<td>15.75</td>
</tr>
<tr>
<td>Null</td>
<td>41</td>
<td>41</td>
<td>0</td>
<td>1.87</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>211</td>
<td>277</td>
<td>66</td>
<td>10.96</td>
</tr>
<tr>
<td>Powerup</td>
<td>51</td>
<td>99</td>
<td>48</td>
<td>6.32</td>
</tr>
<tr>
<td>RadioCountToLeds</td>
<td>183</td>
<td>244</td>
<td>61</td>
<td>9.85</td>
</tr>
<tr>
<td>RadioSenseToLeds</td>
<td>211</td>
<td>277</td>
<td>66</td>
<td>10.80</td>
</tr>
<tr>
<td>Sense</td>
<td>107</td>
<td>159</td>
<td>52</td>
<td>7.88</td>
</tr>
</tbody>
</table>

The time required to identify generic components, copy their corresponding ASTs, update the ASTs by substituting actual parameters for formal parameters, remove the generic markers (e.g., the generic keyword, parameters lists) from the ASTs, and appropriately update all configurations using those components (to resolve references).

A summary of the experimental results is shown in Table 4.6. The File Count column corresponds to the number of files scanned. The Inst. Count column corresponds to the total number of ASTs resulting from the call to instantiateGenerics(). The New Files column includes the difference between the Inst. Count column and the File Count column and corresponds to the number of new ASTs introduced by the instantiation process. Finally, the Time column corresponds to the time required to perform the instantiation. The table shows that MultihopOscilloscope, with 81 newly introduced ASTs, is the application with the largest number of generic components. The identification and instantiation of those ASTs took approximately 18 seconds. On the other hand, Null made use of no generic components and the process took less than 2 seconds. These values indicate that nAIT’s instantiation time is proportional to the number of generic components encountered. The worst-case of 18 seconds makes the toolkit suitable for the development of both interactive and batch-based software engineering tools (O2.2).
### Table 4.7: Interface Source File AST Sizes

<table>
<thead>
<tr>
<th>Application</th>
<th>Files</th>
<th>Min</th>
<th>Max</th>
<th>Median</th>
<th>Mean</th>
<th>Stddev</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseStation</td>
<td>73</td>
<td>1,080</td>
<td>54,500</td>
<td>4,440</td>
<td>5,908</td>
<td>7,531</td>
<td>431,308</td>
</tr>
<tr>
<td>Blink</td>
<td>20</td>
<td>1,064</td>
<td>11,500</td>
<td>5,212</td>
<td>5,239</td>
<td>3,738</td>
<td>104,784</td>
</tr>
<tr>
<td>MViz</td>
<td>89</td>
<td>1,064</td>
<td>54,500</td>
<td>4,472</td>
<td>6,065</td>
<td>7,166</td>
<td>539,872</td>
</tr>
<tr>
<td>MultihopOscilloscope</td>
<td>96</td>
<td>1,096</td>
<td>54,500</td>
<td>4,508</td>
<td>6,159</td>
<td>7,035</td>
<td>591,302</td>
</tr>
<tr>
<td>MultihopOscilloscopeLqi</td>
<td>91</td>
<td>1,104</td>
<td>54,500</td>
<td>4,520</td>
<td>5,925</td>
<td>6,981</td>
<td>539,232</td>
</tr>
<tr>
<td>Null</td>
<td>13</td>
<td>1,064</td>
<td>10,500</td>
<td>2,136</td>
<td>3,922</td>
<td>3,252</td>
<td>50,988</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>73</td>
<td>1,080</td>
<td>54,500</td>
<td>4,496</td>
<td>6,212</td>
<td>7,651</td>
<td>453,476</td>
</tr>
<tr>
<td>Powerup</td>
<td>16</td>
<td>1,072</td>
<td>11,500</td>
<td>4,264</td>
<td>5,018</td>
<td>3,788</td>
<td>80,292</td>
</tr>
<tr>
<td>RadioCountToLeds</td>
<td>65</td>
<td>1,088</td>
<td>54,500</td>
<td>4,504</td>
<td>6,202</td>
<td>7,906</td>
<td>403,140</td>
</tr>
<tr>
<td>RadioSenseToLeds</td>
<td>73</td>
<td>1,088</td>
<td>54,500</td>
<td>4,504</td>
<td>6,221</td>
<td>7,649</td>
<td>454,164</td>
</tr>
<tr>
<td>Sense</td>
<td>35</td>
<td>1,064</td>
<td>16,500</td>
<td>3,944</td>
<td>5,053</td>
<td>3,904</td>
<td>176,872</td>
</tr>
</tbody>
</table>

#### 4.3.3 Space Requirements

Next we consider the space requirements associated with using nAIT. The following paragraphs detail our analysis of the space required to load programs using the toolkit. We consider the results in terms of (1) interfaces, (2) components, (3), modules, and (4) header files. We then compare the median AST size required for each file type. The results include the sizes of ASTs corresponding to complete source files. There are 159 classes for AST nodes. The minimum size of a corresponding object is 24 bytes and the maximum is 48 bytes. The mean size is 25.66 bytes and the standard deviation is 4.12. The tool used to collect this information limits the accuracy of results greater than 9000 bytes to ±500 bytes\(^\text{12}\).

**Interface Space Requirements.** Table 4.7 summarizes the space requirements associated with ASTs corresponding to nesC interfaces. The *Files* column captures the number of files that contain interfaces in the respective applications. The other columns have the obvious meanings. As shown in the table, the minimum AST size for an interface ranges between 1064 bytes and 1104 bytes. The maximum AST size ranges between 10,500 and 54,500 bytes. The minimum and maximum values are shared among several of the applications, resulting from the fact that the corresponding interfaces are core TinyOS interfaces common to many of the applications. The minimum size of 1064 bytes, shared

---

\(^\text{12}\)Using JProfiler, results over 9000 bytes are displayed in terms of kilobytes. We therefore use the midpoint of the displayed kilobyte range (e.g., 10,500 is used for 10k).
by 4 of the applications, corresponds to the file \$TOSDIR/interfaces/Boot.nc, an interface that defines only a single entry, namely event void booted(). Similarly, the maximum size of 54,500 bytes, shared by 7 of the applications, corresponds to the file \$TOSDIR/chips/msp430/usart/HplMsp430I2C.nc, an interface that defines 59 commands.

There is significant variability in the sizes of the interface ASTs, as indicated by the wide ranges between the minimum and maximum values and by the large standard deviations. We therefore consider not only the mean AST size, but also the median. In MultihopOscilloscope, the application with the largest number of interfaces, the total number of bytes used to store interface ASTs is 591,302 (approximately 577kB). Multiplying the median AST size by the total number of ASTs results in 591,302.4 bytes (again, approximately 577kB), while multiplying the mean AST size by the total number of ASTs results in 432,768 bytes (approximately 423 kB). This shows that the median value does, in fact, provide a better estimate in this case. We believe that 577kB is an acceptable amount of memory resources to dedicate to the representation of all interfaces within an application, enabling users to conduct analysis and instrumentation tasks on desktop-class hardware (O2.3).

Configuration Space Requirements. Table 4.8 summarizes the space requirements associated with ASTs corresponding to nesC configurations. As shown in the table, the minimum size of a configuration AST ranges from 2008 bytes to 2376 bytes. The maximum size ranges from 68,500 bytes to 132,500 bytes. As with interfaces, there are a number of applications that share the same minimum and maximum sizes, again resulting from the fact that both the smallest and largest configurations in the programs are part of the common TinyOS library. The minimum size of 2344 bytes, shared by 4 of the applications, corresponds to the file \$TOSDIR/platforms/telosb/MoteClockC.nc, a configuration that provides 1 interface and includes 2 component references and 1 wiring statement. The maximum size of 133,500 bytes, shared by 6 of the applications, corresponds to the file \$TOSDIR/chips/msp430/pins/HplMsp430GeneralIOC.nc, a configuration that provides 74 interfaces and includes 48 component references and 74 wiring statements.
There is significant variability in the sizes of the configuration ASTs, as indicated by the wide ranges between the minimum and maximum values and, by the large standard deviations. We therefore consider not only the mean AST size, but also the median. In MultihopOscilloscope, the application with the largest number of configurations, the total number of bytes used to store configuration ASTs is 1,001,744 bytes (approximately 978kB). Multiplying the median AST size by the total number of ASTs results in 1,001,743.6 bytes (again, approximately 978kB), while multiplying the mean AST size by the total number of ASTs results in 1,407,045.9 bytes (approximately 1.34MB). As with interfaces, the median value provides a better estimate in this case. We believe that 978kB is an acceptable amount of memory resources to dedicate to the representation of all configurations within an application, enabling users to conduct analysis and instrumentation tasks on desktop-class hardware (O2.3).

**Module Space Requirements.** Table 4.9 summarizes the space requirements associated with ASTs corresponding to nesC configurations. The minimum size of a module AST ranges from 1736 bytes to 6968 bytes. The maximum size ranges from 71,500 bytes to 389,500 bytes. The applications do not share common minimum module AST sizes, suggesting that the smallest modules are not likely to be part of the TinyOS library. In Null, for instance, the minimum size of 1736 bytes corresponds to the application-level module NullC, a module that uses 1 interface and implements 1 empty event. Similarly,
<table>
<thead>
<tr>
<th>Application</th>
<th>Files</th>
<th>Min</th>
<th>Max</th>
<th>Median</th>
<th>Mean</th>
<th>Stddev</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseStation</td>
<td>50</td>
<td>6,968</td>
<td>256,500</td>
<td>34,000.00</td>
<td>60,854.16</td>
<td>59,198.91</td>
<td>3,042,708</td>
</tr>
<tr>
<td>Blink</td>
<td>21</td>
<td>5,616</td>
<td>71,500</td>
<td>28,500.00</td>
<td>29,719.24</td>
<td>18,735.46</td>
<td>624,104</td>
</tr>
<tr>
<td>MViz</td>
<td>63</td>
<td>2,816</td>
<td>389,500</td>
<td>33,500.00</td>
<td>67,630.29</td>
<td>78,173.27</td>
<td>4,260,708</td>
</tr>
<tr>
<td>MultihopOscilloscope</td>
<td>72</td>
<td>2,848</td>
<td>389,500</td>
<td>37,000.00</td>
<td>70,835.06</td>
<td>79,563.37</td>
<td>5,100,124</td>
</tr>
<tr>
<td>MultihopOscilloscopeLqi</td>
<td>72</td>
<td>4,832</td>
<td>295,500</td>
<td>35,000.00</td>
<td>63,997.22</td>
<td>64,831.98</td>
<td>4,607,800</td>
</tr>
<tr>
<td>Null</td>
<td>11</td>
<td>1,736</td>
<td>71,500</td>
<td>27,500.00</td>
<td>29,289.82</td>
<td>21,939.52</td>
<td>322,188</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>55</td>
<td>4,808</td>
<td>295,500</td>
<td>34,500.00</td>
<td>58,563.05</td>
<td>62,295.55</td>
<td>3,220,968</td>
</tr>
<tr>
<td>Powerup</td>
<td>14</td>
<td>2,808</td>
<td>71,500</td>
<td>30,000.00</td>
<td>29,483.43</td>
<td>19,909.64</td>
<td>412,768</td>
</tr>
<tr>
<td>RadioCountToLeds</td>
<td>47</td>
<td>5,640</td>
<td>256,500</td>
<td>29,500.00</td>
<td>52,535.49</td>
<td>53,206.10</td>
<td>2,469,168</td>
</tr>
<tr>
<td>RadioSenseToLeds</td>
<td>55</td>
<td>4,816</td>
<td>295,500</td>
<td>34,500.00</td>
<td>58,072.87</td>
<td>61,772.53</td>
<td>3,194,008</td>
</tr>
<tr>
<td>Sense</td>
<td>30</td>
<td>4,800</td>
<td>295,500</td>
<td>31,000.00</td>
<td>45,580.13</td>
<td>56,695.85</td>
<td>1,367,404</td>
</tr>
</tbody>
</table>

Table 4.9: Module Source File AST Sizes

in Powerup, the minimum size of 2808 bytes corresponds to the application-level module PowerupC, a module that uses two interfaces and implements 1 event that makes 1 function call. The maximum sizes do, however, suggest that the applications share a set of large core TinyOS modules. For example, the maximum size of 295,500 bytes, shared by 4 of the applications, corresponds to $\$TOSDIR/chips/msp430/adc12/Msp430Adc12ImpIP.nc$, a module that provides 5 interfaces, uses 15 interfaces, and contains 1 enumeration with 9 elements, 6 state variables, and 16 non-empty functions.

Again, because of the significant variability in AST sizes, indicated by the wide ranges between the minimum and maximum values and by the large standard deviations, we consider both the mean and the median values. In MultihopOscilloscope, the application with the largest number of modules, the total number of bytes used to store module ASTs is 5,100,124 bytes (approximately 4.9MB). Multiplying the median AST size by the total number of ASTs results in 5,100,124 bytes (again, approximately 4.9MB), while multiplying the mean AST size by the total number of ASTs results in 5,728,562.64 bytes (approximately 5.5MB). This shows that the median value provides a better estimate in this case. We believe that 4.9MB is an acceptable amount of memory resources to dedicate to the representation of all modules within an application, enabling users to conduct analysis and instrumentation tasks on desktop-class hardware (O2.3).
<table>
<thead>
<tr>
<th>Application</th>
<th>Files</th>
<th>Min</th>
<th>Max</th>
<th>Median</th>
<th>Mean</th>
<th>Stddev</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseStation</td>
<td>13</td>
<td>680</td>
<td>1,271,500</td>
<td>6,824.00</td>
<td>169,550.15</td>
<td>337,150.78</td>
<td>2,204,152</td>
</tr>
<tr>
<td>Blink</td>
<td>9</td>
<td>2,992</td>
<td>1,271,500</td>
<td>24,500.00</td>
<td>229,219.11</td>
<td>389,047.11</td>
<td>2,062,972</td>
</tr>
<tr>
<td>MViz</td>
<td>20</td>
<td>680</td>
<td>1,271,500</td>
<td>9,500.00</td>
<td>116,577.80</td>
<td>284,415.30</td>
<td>2,331,556</td>
</tr>
<tr>
<td>MultihopOscilloscope</td>
<td>21</td>
<td>680</td>
<td>1,271,500</td>
<td>9,500.00</td>
<td>109,921.90</td>
<td>275,999.39</td>
<td>2,308,360</td>
</tr>
<tr>
<td>MultihopOscilloscopeLqi</td>
<td>18</td>
<td>680</td>
<td>1,271,500</td>
<td>11,162.00</td>
<td>126,603.33</td>
<td>294,825.96</td>
<td>2,278,860</td>
</tr>
<tr>
<td>Null</td>
<td>8</td>
<td>2,992</td>
<td>1,271,500</td>
<td>64,500.00</td>
<td>257,298.50</td>
<td>403,957.04</td>
<td>2,058,388</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>15</td>
<td>680</td>
<td>1,271,500</td>
<td>6,824.00</td>
<td>149,350.67</td>
<td>318,105.00</td>
<td>2,240,260</td>
</tr>
<tr>
<td>Powerup</td>
<td>9</td>
<td>2,992</td>
<td>1,271,500</td>
<td>24,500.00</td>
<td>229,219.11</td>
<td>389,047.11</td>
<td>2,062,972</td>
</tr>
<tr>
<td>RadioCountToLeds</td>
<td>14</td>
<td>680</td>
<td>1,271,500</td>
<td>6,716.00</td>
<td>157,575.43</td>
<td>327,742.95</td>
<td>2,206,056</td>
</tr>
<tr>
<td>RadioSenseToLeds</td>
<td>15</td>
<td>680</td>
<td>1,271,500</td>
<td>6,824.00</td>
<td>149,199.73</td>
<td>318,174.17</td>
<td>2,237,996</td>
</tr>
<tr>
<td>Sense</td>
<td>11</td>
<td>680</td>
<td>1,271,500</td>
<td>24,500.00</td>
<td>190,468.36</td>
<td>361,439.55</td>
<td>2,095,152</td>
</tr>
</tbody>
</table>

Table 4.10: Header File AST Sizes (nonempty)

**Header File Space Requirements.** Table 4.10 summarizes the space requirements associated with ASTs corresponding to header files used by nesC programs. The minimum size ranges from 680 bytes to 2992 bytes. The maximum size is consistently 1,271,500 bytes. As with other file types, the minimum and maximum values are shared across several programs, resulting from the fact that the smallest and largest header files are included with the core TinyOS source base. The minimum size of 680 bytes, shared by 8 of the applications, corresponds to the file $TOSDIR/types/Resource.h$, a header file that contains a single typedef. The maximum size of 1,271,500, shared by all the applications, corresponds to the file $TOSDIR/platforms/telosb/hardware.h$, a header file that contains 44 macros, each of which expands to the definition of 8 functions (a total of 352 functions), 9 additional functions, and 1 enumeration with 3 elements.

Each of the applications contains relatively few header files and this single large file dominates the mean file size. As before, because of the significant variability in AST size, indicated by the wide ranges between the minimum and maximum values and by the large standard deviations, we consider both the mean and the median values. In MultihopOscilloscope, the application with the largest number of header files, the total number of bytes used to store header ASTs is 2,308,360 (approximately 2.2MB). Multiplying the median AST size by the total number of ASTs results in 2,308,359.9 bytes (again, approximately 2.2MB), while multiplying the mean AST size by the total number of ASTs results...
in 5,795,987.19 bytes (approximately 5.5MB). As with the other AST types, the median value provides a better estimate of file size in this case. We believe that 2.2MB is an acceptable amount of memory resources to dedicate to the representation of all the header files within an application, enabling users to conduct analysis and instrumentation tasks on desktop-class hardware (O2.3).

Finally, Figure 4.6 compares the median AST sizes of each file type across all applications. Clearly the size of ASTs corresponding to modules is almost always the largest. Intuitively, this is because all implementation code for nesC applications are contained within modules. The one exception, shown in the figure, is the Null application, where the median size of the headers exceeds the median size of the modules. This “do-nothing” application is hardly representative of normal nesC applications, as it consists of very few source files.

Our results show that the resources required by nAIT make it well-suited for source analysis and instrumentation tasks on desktop-class hardware (O2.3). Our observed worst-case parse time was a mere 2.26 seconds. Also, our worst-case sample traversal time for identifying module, function, and function call relations was 1.63 seconds. The maximum time of 18.4 seconds to instantiate generic components was more significant, but still not
prohibitive. Similarly, the largest application in our test suite, MultihopOscilloscope, consumes a total of 9,001,530 bytes (approximately 8.6MB) for all the ASTs in the system. Today’s desktop-class machines with gigabytes of main memory can easily support this program and programs that are significantly larger.

4.4 Research Contributions

We have described the design and implementation of nAIT, our implementation of a source analysis and instrumentation framework for nesC — Contribution 2 of this dissertation. We identified design desiderata and showed how our implementation satisfies those desiderata. We showed that our source-based approach enables platform-independent analysis and instrumentation activities. We showed that our API simplifies the development of language processing tools and source-level instrumentation tools. We also showed that our AST Explorer aids developers in understanding the structure of the in-memory program representation maintained by the API. We presented a use-case scenario that leverages the services provided by the API to analyze a nesC source base to identify state variables within modules and to instrument the source base to monitor state predicates at runtime. (Our third contribution, detailed in Chapter 5, also serves as a use-case for this toolkit.) Finally, we evaluated the contribution in terms of its suitability of purpose, as well as its runtime and memory requirements.

Prior to this work, there was no suitable general-purpose framework for analyzing and instrumenting nesC applications. The nesC platform is fundamentally different from existing imperative programming languages, making existing frameworks inapplicable. As a result, analysis and instrumentation activities were performed manually by developers. These manual changes were tedious, time-consuming, and error-prone. Additionally, the lack of such a general-purpose framework limited the number of software engineering tools that were developed to aid in addressing the difficulties associated with the development of embedded network systems.
The Analysis and Instrumentation Framework solves these problems. The framework enables analysis and instrumentation techniques that accommodate the novel features of the nesC language. The framework provides an API that enables users to develop new software engineering tools that support analysis and instrumentation activities. The API enables static analysis of application source code (e.g., the construction of static system call graphs). It also provides services for traversing, generating, and modifying portions of ASTs corresponding to nesC source files programmatically. These services make the development of new software engineering tools quick and easy. This contribution has, and will continue to improve the development of embedded network systems.
Chapter 5

Control-Flow Visualization Framework

In Chapter 1 we identified the problem that there are no techniques to visualize nesC control-flow. In this chapter, we describe our third contribution – a control-flow visualization framework for nesC programs. The framework supports the visualization of both static system call graphs and dynamic trace data and enables developers to more easily understand the flow of control through nesC programs. We refer to our implementation of this framework as the Visualization Toolkit. The Visualization Toolkit realizes the framework in “stand-alone” mode and enables control-flow visualization for individual devices. We have expanded this toolkit and integrated it with the NESTbed system described in Chapter 3 to produce the Visualization Testbed. The Visualization Testbed expands the set of services provided by the Visualization Toolkit to enable users to collect, identify correspondences between, and visualize network send and receive events. The visualization enables developers to more easily understand the causal relationships between distributed events. This chapter is based on [28, 29].

Design Desiderata. The framework design goals are as follows. First, the framework must support static control-flow visualization. The static visualizations will enable developers to easily understand the set of possible execution paths through target systems.
and will serve as a guide to developers as they identify portions of their programs to study more carefully. Second, the framework must support *dynamic* control-flow visualization. The dynamic visualization will enable developers to more easily understand the *actual* execution paths through target systems. The objective of these visualizations is to aid developers in understanding the control-flow in reactive, event-based systems. Third, the framework must support static and dynamic event filtering. Static event filters will enable developers to select events of interest prior to system execution. Dynamic event filtering will enable developers to further refine the events of interest after system execution. The objective of the filtering is to enable users to “trim” a tremendous amount of potential control-flow data to include only the information of interest. Finally, the framework must support visualization and filtering activities across multiple devices. The inter-device visualizations must enable users to correlate send and receive events across the wireless radio. These visualizations will enable users to more easily understand the causal relationships between distributed events.

The following sections discuss the implementation and evaluation of the Visualization Toolkit. Section 5.1 presents the software components of the Visualization Toolkit. Section 5.2 describes how the Visualization Toolkit has been expanded and integrated with the NESTbed to produce the Visualization Testbed. Section 5.3 presents three use-case scenarios, two that illustrate the use and benefits of the Visualization Toolkit, and one that illustrates the use and benefits of the Visualization Testbed. Section 5.4 presents an evaluation of both the Visualization Toolkit and Visualization Testbed in terms of their suitability of purpose, the rate at which they can capture dynamic trace events, and the resource overhead introduced by their application. Finally, Section 5.5 summarizes our research contributions in the area of embedded network system visualization.
5.1 Visualization Toolkit

In this section, we present the Visualization Toolkit, a platform-neutral toolkit for TinyOS 2.0 to aid in program comprehension. This is, to the best of our knowledge, the first toolkit of its kind for any embedded network platform. Two modes of operation are supported. The first is focused on static application structure and, consequently, on potential execution paths. The output in this mode consists of an annotated system call graph corresponding to a user’s source base. The second — and more interesting — mode of operation is focused on dynamic application structure and, consequently, on actual execution paths. The output in this mode is an annotated UML sequence diagram corresponding to the behavior of a single application run.

The toolkit architecture consists of four components; these components are the focus of this section. First, we describe a component constructed using nAIT to insert logging probes within a source base to capture the program actions of interest to a developer. Second, we describe a lightweight service for recording TinyOS execution events. Third, we describe a tool to extract logged execution events and to reconstruct the underlying runtime trace. Finally, we describe two visual front-ends corresponding to the static and dynamic views introduced above.

The steps involved in applying the Visualization Toolkit are summarized in Figure 5.1. The first step is to analyze and transform the source base of the system under test. The output of this step is an instantiated source base free of generic types and a collection of
metadata detailing program symbols and function calling relationships. In the simple case, this metadata can be used to generate an annotated call graph for the system under test, enabling users to understand the potential flow of control. In the more interesting case, the metadata is used as input to a second step, in which a developer selects a static set of functions to be traced. The selected functions are used to guide the insertion of probes that record enter and exit events on the functions of interest. The output of this step is an instrumented source base and a symbol map that associates numeric identifiers captured in a trace with their meaningful symbol names. The instrumented system is then compiled and executed on the target hardware (or simulation) platform. The resulting execution trace must then be extracted from the hosting device. The extracted trace is reconstructed using the symbol map to substitute symbol names for the numeric identifiers it contains. The reconstructed trace can then be used to generate a sequence diagram that describes the program run and helps developers understand how control flowed through the system. The individual components of the toolkit that enable this process are described in the following subsections.

5.1.1 Probe Selector

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

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

The second transformation involves injecting logging probes at the entry and exit points of the selected actions. The basic instrumentation procedure is illustrated by the code fragments shown in Listings 5.1 and 5.2. The first listing shows a simple nesC command prior to instrumentation; the second shows equivalent instrumented code. The body of each instrumented action is wrapped within an anonymous block, and the enter event is recorded before the block. This allows probes to be injected before variable declarations (which might involve function calls). Similarly, each exit point is wrapped within a statement-expression to capture exit events before the action terminates. In general, multiple exit points may need to be instrumented. (Recall that nAIT normalizes nesC code by injecting explicit return statements. This normalization simplifies this instrumentation task.)

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

### 5.1.2 Event Recording Service

The *Event Recording Service* is implemented as a single component, `TraceRecorderC`, which serves as a thin wrapper over the standard log storage component provided by TinyOS (`LogStorageC`). The component provides a simple interface, `Trace`, which defines commands to log *entry* and *exit* events. (The component is initialized at system startup.) Internally, the component implements a dual buffering strategy. A *current* buffer is used to cache program events logged to the trace. When the current buffer becomes full, the second buffer is swapped into its place while the first buffer is flushed. This cyclic process repeats to prevent missed events during log storage updates. We note that in our testing of `TraceRecorderC`, 70 entries of 3 bytes each is close to the maximum buffer size that can be consistently written to log storage without introducing bit errors.

### 5.1.3 Trace Extractor / Collector

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

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

5.1.5 Sequence Diagram Generator

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

To enable users to distinguish between commands, events, tasks, and module functions, we use the coloring scheme summarized in Table 5.1. Orange, purple, blue, and green activation rectangles are associated with commands, events, tasks, and module functions, respectively. Similarly, to enable users to distinguish between synchronous and asynchronous actions, we use the edge coloring scheme summarized in the table. Black and red edges are used to represent calls to synchronous and asynchronous actions, respectively. Blue edges are used to represent calls to local functions, which can be invoked from either a
synchronous or asynchronous context. (We note that edge color is based on static signature data; determining the runtime context of a call will typically require probes outside the system’s instrumentation set.) We shall see that this coloring scheme improves the utility of the sequence diagram notation in supporting developers’ understanding of system behavior.

### 5.1.6 Callers, Callees, and Visualization Uncertainty

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

Consider a program $P$, for which some subset of the program actions will be instrumented and traced. We refer to this set as the instrumentation set of $P$. A portion of the call graph for $P$ is shown in Figure 5.2. Each circle represents a program action; shaded circles denote actions in the instrumentation set. Without a loss of generality, assume that $S()$ is the closest instrumented action between $S()$ and $D()$.

Consider generating a sequence diagram based on a trace of $P$. In particular, consider an element of the trace, $D_b$, corresponding to an enter event on action $D()$. The basic visualization task is to determine which action invoked $D()$ — either directly, or through a call chain outside the instrumentation set of $P$. It is important to observe that a simple projection of the runtime stack at $D_b$ can be constructed by tracing the enter and

<table>
<thead>
<tr>
<th>Type</th>
<th>Color</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calls / Returns</td>
<td>Black</td>
<td>Synchronous</td>
</tr>
<tr>
<td></td>
<td>Red</td>
<td>Asynchronous</td>
</tr>
<tr>
<td></td>
<td>Blue</td>
<td>Module functions</td>
</tr>
<tr>
<td>Activations</td>
<td>Orange</td>
<td>Commands</td>
</tr>
<tr>
<td></td>
<td>Purple</td>
<td>Events</td>
</tr>
<tr>
<td></td>
<td>Blue</td>
<td>Tasks</td>
</tr>
<tr>
<td></td>
<td>Green</td>
<td>Module functions</td>
</tr>
</tbody>
</table>

Table 5.1: Sequence Diagram Color Mapping
exit events recorded on the trace. The projection includes (conceptual) activation records only for those actions in the instrumentation set.

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

Now assume that D() is asynchronous\footnote{We include module-level functions in this set because they may be invoked from an asynchronous context.}. In this case, the path to D() may have come from either a synchronous or asynchronous context. In assigning a caller to D(), the system again examines the runtime stack at $D_b$, excluding actions that could not have lead to D(). If there are no suitable candidates, the System action is assigned as D()’s caller.
Otherwise, the topmost candidate—synchronous or asynchronous—is selected; assume that this action is \( S() \). In some cases, this assignment is incorrect.

Under what conditions could \( S() \) be the topmost action on the runtime stack (at \( D_b \)) according to the captured trace, but not be responsible for the call to \( D() \)? There is only one possibility: The execution path through \( S() \) must have been interrupted by a new call chain that reached \( D() \). This chain must satisfy two key properties. First, it must have originated from an asynchronous context because synchronous paths are non-preemptive. Second, it must flow along an uninstrumented path in \( P \); otherwise \( S() \) would not be the topmost candidate recorded on the captured trace. Again consider Figure 5.2. An asynchronous flow through \( Y() \) to \( D() \) could interrupt a call chain from \( S() \) to \( D() \). The visualization system would incorrectly associate the first call to \( D() \) with \( S() \) and the second call to \( D() \) with \( Y() \). Note that this association error would not occur if a preemptive path through \( X() \) reached \( D() \) because \( X() \) is included in the instrumentation set of \( P \). Hence, in general, an association error will occur if an asynchronous flow reaches an asynchronous action that is reachable from the topmost action reflected on the recorded trace.

The frequency of association errors depends on the application under test, as well as the corresponding instrumentation set. In practice, however, the occurrence rate seems likely to be rare. The risk can also be reduced by introducing additional instrumentation points to disambiguate asynchronous call chains. Still, a limited degree of uncertainty exists and it would be preferable to reduce or eliminate it.

5.2 Expansion and Testbed Integration

In this section we discuss a significant extension to the Visualization Toolkit described in the previous section. Whereas the Visualization Toolkit focuses exclusively on local program behavior, the focus of this section is on understanding distributed program behavior. We present a distributed visualization approach that is integrated with the Interactive Testing Framework described in Chapter 3. The product of this expansion and
integration is the Visualizatıon Testbed. The Visualizatıon Testbed enables analysis, instrumentatıon, and visualizatıon of embedded programs\textsuperscript{2}.

In describing the design and implementation of the Visualizatıon Testbed, it is useful to follow a process-oriented approach. The key steps involved in visualizatıng the behavior of an application using the testbed are summarized in Figure 5.3. (We omit testbed activities irrelevant to program visualizatıon; see Chapter 3 for a complete description.) Blue ovals represent client-side activities, and yellow ovals represent server-side activities. The process begins with program selection (1), which involves supplying a client-side source path. The identified materials are automatically archived and uploaded to the NESTbed server for storage. At the server, the Makefile included in the archive is parsed to identify the application’s top-level configuration. The path to this file is passed as input to nAIT to initiate a full system parse and an analysis of the resulting parse trees (2).

\textsuperscript{2}The Visualizatıon Testbed is the product of a collaborative effort between the author, Sravanthi Dandamudi, and Sally K. Wahba. The author was responsible for the development of the original visualization front-end (discussed in Section 5.1), as well as for the integration of the distributed visualization with the NESTbed. Dandamudi and Wahba were responsible for modifying the visualization front-end to include the send and receive events and for implementing the logic to “link” those visualizations. The work presented here focuses on the author’s contributions.
The analysis phase produces a property set that records the identified program modules, symbols, calling relationships, and component wiring relationships. This information is stored as part of the active testbed project and used to support two visualization paths involving call graph and sequence diagram construction. Both paths begin with (elements of) the property set being transferred back to the client to support visualization filtering (3). This visualization filtering contributes to achieving our design goal of enabling users to visualize only the data in which they are interested through static event filtering. The basic mechanism is identical in each case. The client provides an interface for selecting the program actions to include in the generated visualization. In the case of call graph generation (4), the resulting filter set can be applied against the property set to produce a call graph focused about the selected actions of interest. The scenario is more involved in the case of sequence diagram construction.

In this case, the filter set is transmitted to the server where it is used to guide source instrumentation (5). The symbol map used to assign numeric identifiers to the selected actions is also generated and transmitted. At the server, nAIT is used to include the Trace Recording Service and to inject logging calls that capture entry and exit events on actions within the filter set. Radio transmission and reception events are automatically traced. In addition, the TinyOS messaging libraries are modified to include the sender’s address as part of each outgoing message, as well as a sequence number\(^3\). This additional message data is stored as part of the log entries associated with send and receive events. This will later enable the Visualization Testbed to match corresponding events across nodes. Finally, the instrumented system is compiled (6) and installed (7) on the test network.

After the system has executed for the desired duration, trace extraction (8) is required to collect the data logged at each of the nodes executing the instrumented application. The process is controlled through the dynamic filtering (10) options provided by the client interface. As before, the interface provides controls for selecting the actions to be included in the generated visualization; only traced actions are available for selection. In addition,\(^3\)

\(^3\)The messaging behavior is implemented using conditional compilation in the server-side TinyOS libraries because this behavior remains constant across instrumented systems.
controls are provided for selecting the nodes to include and the trace window to be visualized. The window is specified by an offset (into the trace) and a length, both in terms of logged events. These features enable users to apply dynamic event filtering to visualize only the data in which they are interested. When a node is selected, the client checks its local trace cache to determine whether the corresponding trace has been received from the NESTbed server. If not, the client instructs the server to install the Trace Extractor on the device. The Trace Collector is automatically executed (at the server) to receive the trace data. The data is then transmitted to the client, where trace reassembly is performed using the symbol map produced earlier. When all of the required trace data has been extracted and reassembled, sequence diagram generation proceeds. We note that new dynamic filters can be applied without additional round-trip exchanges with the NESTbed server.

The Visualization Testbed presents users with UML sequence diagrams, each providing a local view of the associated node’s behavior. In addition to the local actions recorded in the diagrams, send and receive events are displayed. Buttons are associated with these events, enabling users to navigate between a send event and its corresponding receive event(s), or vice-versa. If a send event is a broadcast message, multiple recipients may be available from which to choose. If a message is lost, no associated recipient will be available. These “linked” views enable users to reason about the causal relationships between distributed events and provide insight into the behavior of the system in the presence of lost messages.

5.3 System Use-Cases

At this point it is useful to consider three scenarios that illustrate the use of the contributions presented in this chapter. The first two scenarios illustrate the use of the Visualization Toolkit and the benefits that it provides. The scenarios involve standard application examples included as part of the TinyOS 2.0 distribution: Blink and Radio-
CountToLeds. Despite their lack of surface complexity, these applications are rich with interesting behavior. The final scenario illustrates the steps involved in using the Visualization Testbed, as well as the program comprehension benefits that it affords. The scenario involves RadioSenseToLeds, another example included as part of the TinyOS 2.0 distribution. Our goal is to demonstrate the testbed’s utility in managing the non-determinism associated with (1) distributed, concurrent behavior, (2) lossy message delivery, and (3) event-based execution.

5.3.1 Scenario 1: Blink

The scenario begins with a developer interested in using the Visualization Toolkit to investigate the timing behavior of Blink. As a starting point, she may choose to view the system call graph by invoking the Call Graph Generator, passing the top-level configuration file, BlinkAppC.nc, as argument. After selecting VirtualizeTimerC.fireTimers() as the focal point of the visualization, the view shown in Figure 5.4 is displayed. At a glance, the developer might take an interest in the fact that all system timers are being dispatched from fireTimers().

To investigate this behavior further, she may choose to visualize the execution of Blink. She first selects the program actions to be traced using the Probe Selector shown in Figure 5.5, again passing BlinkAppC.nc as argument. In the figure, she has already selected several actions. When the selection process is complete, the instrumented source base is generated, and the corresponding symbol map is exported for later use. The system is then compiled and installed.

To collect the resulting trace data, the Trace Extractor is installed on the target device, and the Trace Collector is executed to reconstruct the runtime trace. The symbol map is passed as argument at startup. Finally, the reconstructed trace is passed to the Sequence Diagram Generator to produce the diagram shown in Figure 5.6.

---

4The figures in this section were generated using the Tmote Sky platform from Moteiv [78]. All figures have been trimmed and condensed for the sake of presentation. Solid horizontal breaks denote event omissions.
The diagram captures a canonical example of *device virtualization* in TinyOS. The first execution sequence begins with an asynchronous event, `Alarm.fired()`, signaled on `Alarm-ToTimerC`. The event originates from an *actual* clock source. (The event source is designated as `System` because the actual signaling action is outside the instrumentation set of `Blink`.) This event posts a task, `fired()`, which is the next action executed in the sequence. The task signals `TimerFrom.fired()` on `VirtualizeTimerC`, which in turn invokes a local dispatching function, `fireTimers()`. The dispatching function signals `fired()` on all pending virtual timer instances. Finally, these events trigger the main `BlinkC` module to invoke LED toggle functions on `LedsP`. In the first series of program actions, all the virtual timers used by `Blink` are pending, so each `fired()` event is signaled. In the second series, only two of the virtual timers are pending (i.e., `Timer1` and `Timer0`); hence there are only two `fired()` events reflected in the diagram. We note that it can be difficult to reason about the behavior of a virtualized device based on manual inspection of the program source code. By contrast, the behavior is clear from the sequence diagram.

The third execution sequence is also interesting and reveals a minor modification of `Blink` introduced for testing purposes. Specifically, `Blink` was modified to include an asynchronous event triggered when the *user button* is clicked on the hosting device. As
Figure 5.6: Sequence Diagram for Blink
shown in the diagram, this event *interrupted* the (synchronous) Timer3.fired() event. While the example is obviously contrived, it is representative of a larger class of behaviors — behaviors that can be difficult to understand without a visualization tool.

5.3.2 Scenario 2: RadioCountToLeds

The second scenario is focused on a developer interested in using the Visualization Toolkit to investigate the runtime behavior of RadioCountToLeds. We omit the individual steps in the visualization process and skip to the generated sequence diagram shown in Figure 5.7.

Once again the generated diagram captures interesting behavior that could otherwise be difficult to understand. In particular, the diagram illustrates *active message filtering* and *dispatch* in the CC2420 radio stack. Three call chains are captured, each triggered by the receipt of a message. Each chain begins with the execution of receiveDone_task(), a task implemented by CC2420ReceiveP. This task is responsible for populating elements of the message header before signaling the SubReceive.receive() event on UniqueReceiveP. Within UniqueReceiveP, hasSeen() is invoked to determine if the message is a duplicate of a previous message. The behavior illustrated in the figure indicates that the received messages were *not* duplicates; SubReceive.receive() is consequently signaled on CC2420ActiveMessageP in each chain. This event is responsible for filtering messages based on the destination address and performing dispatch based on the active message identifier within the message. AMPacket.isForMe() is invoked to determine whether the message is addressed to the hosting node (or intended for all nodes). In the first call chain, the message is intended for the hosting device, and Receive.receive() is signaled on RadioCountToLedsC, which updates the state of the host’s LEDs based on the content of the message. In the second call chain, AMPacket.isForMe() returns FALSE, and the message is silently discarded. In the final call chain, the message is intended for the hosting device, but the specified active message identifier does not have an associated handler within the application. In this case, the
Figure 5.7: Sequence Diagram for RadioCountToLeds
default handler defined in CC2420ActiveMessageP is executed. We note that this diagram helped us to resolve errors in our own understanding of the radio stack’s behavior\textsuperscript{5}.

5.3.3 Scenario 3: RadioSenseToLeds

The third scenario begins with a developer interested in using the Visualization Testbed to explore the behavior of RadioSenseToLeds. She first constructs a new NESTbed project and a deployment configuration within that project. (Recall from Chapter 3 that the deployment configuration stores image-to-device mappings, power level settings, and symbols to be profiled, as configured by the user.) To populate the project, she chooses to upload a new TinyOS application (RadioSenseToLeds) from her local machine. As part of the dialog used to supply the source path, she chooses to inject logging probes and is presented with the Action Selection window. The window is similar to the Probe Selector window shown in Figure 5.5; we omit an additional screen capture. The window lists the modules included in RadioSenseToLeds, as well as the associated commands, events, tasks, and (private) functions. The selected actions comprise the instrumentation set of RadioSenseToLeds. The set serves as a static filter, bounding the actions captured for visualization.

When the selection process is complete, the instrumented system is compiled by the NESTbed server, and the resulting program image is available for installation. We omit the details related to configuring the deployment, but assume that our user has assigned the program image to nodes 0–9 (adjacent nodes in the first row of the test deployment). At this point, the user may activate the deployment configuration, triggering installation on each configured device. Alternatively, she might choose to inspect a particular node using the NESTbed Network Monitor (Section 3.2.3). In addition to providing activation functions, the window presents a grid-based view of the test network and options for retrieving device configuration details. In particular, a user can choose to view the static call graph

\textsuperscript{5}We were unaware of the automatic address filtering performed by the CC2420. This behavior is immediately obvious from the sequence diagram generated when address recognition is enabled on the radio chip. This feature was disabled to generate the above diagram.
corresponding to the application mapped to a selected device. Assume that our user selects node 0 (configured to execute RadioSenseToLeds). She is then prompted to select program actions about which the generated call graph should be focused. The window is similar to the one shown in Figure 5.5; we omit an additional screen capture.

Call graph generation is typically performed iteratively as the user identifies the event regions of interest and determines the appropriate level of detail to display. After a few iterations, assume that our user has focused the graph about the Read.readDone() event implemented by RadioSenseToLedsC and the readDone() task implemented by AdcP. In this case, she is presented with the call graph shown in Figure 5.8. Notice that the graph displays the action of interest, along with additional context hints (i.e., callers and callees). These hints can be used to adjust the action filter to reveal additional context about the application’s potential runtime behavior — in effect, widening the lens through which the graph is viewed. Our developer might, for example, notice that readDone()...
invokes `AMSend.send()` on `AMQueueEntryP` and choose to explore lower-level radio stack behavior through this call chain.

The more interesting exploration path involves exploring the application’s runtime behavior. To achieve this, our user first activates the current configuration, installing the instrumented image on each device. After allowing the system to execute for some time, she chooses to visualize the behavior of nodes currently executing `RadioSenseToLedsC`. At this point, the user is prompted with the Visualization Filter window shown in Figure 5.9. The window provides options for applying dynamic filters to the collected trace data before the corresponding sequence diagrams are generated. The topmost segment controls the nodes to be included in the visualization. (Only nodes running the selected program are available for inclusion.) In the figure, our user has selected alternating nodes in the top row. The second segment controls the trace window to be visualized. The selection in the figure corresponds, approximately, to the first 1000 events captured in each local trace. The third segment shows the actions and associated modules previously selected for capture. Only those actions dragged into the fourth segment will be included in the generated diagrams.
The system generates one sequence diagram per node and displays the diagrams in a tabbed format, as shown in Figure 5.10a. In the figure, our user has selected the tab corresponding to \emph{mote 0}. (Tab labels denote node identifiers.) The execution behavior is obvious from the diagram: The trace begins with the \texttt{MilliTimer.fired()} event, signaled on \texttt{RadioSenseToLedsC}. In turn, the component invokes the split-phased \texttt{Read.read()} command on \texttt{AdcP} (to retrieve the current internal voltage of the device). After control returns to System, a radio message is received, as indicated by the \emph{open} envelope on the System timeline. \texttt{Receive.receive()} is consequently signaled on \texttt{RadioSenseToLedsC}, which in turn updates the state of the LEDs through \texttt{LedsP} (based on the content of the received message). Next, \texttt{SingleChannel.singleDataReady()} is signaled on \texttt{AdcP}, indicating that the previously requested (voltage) data is now available. \texttt{AdcP} then posts the \texttt{readDone()} task, which, when executed, signals \texttt{Read.readDone()} on \texttt{RadioSenseToLedsC}. The \emph{closed} envelope within the activation rectangle of \texttt{readDone()} indicates that a message was \emph{sent}. The corresponding \texttt{AMSend.sendDone()} event can be seen at the bottom of the diagram.

The envelope icons within the graph are used to navigate between corresponding send and receive events. This navigation facility assists users in correlating distributed behaviors and in understanding the impact of failed transmissions. In the figure, our user has clicked the envelope corresponding to the last send event. The event was a \emph{broadcast} transmission received by nodes 2 and 8 (but not by nodes 4 and 6), as shown in the figure. We assume that she selects \emph{mote 8}. As a result, the view jumps to the corresponding receive event in the sequence diagram for this node. Figure 5.10b shows the resulting display. The target receive event is indicated by the highlighted magnifying glass. From here, our user may trace the behavior of node 8 resulting from the receipt of the message, or return to the original send event by clicking on the magnifying glass. The arrows at the bottom of the display provide additional navigation functions equivalent to the history-based functions provided by standard web browsers.

While the scenario is focused on a simple application, it illustrates the power of the Visualization Testbed in enabling developers to understand the execution possibilities
underlying their networks and the particular execution paths chosen during each run. In short, the Visualization Testbed provides a novel and effective infrastructure for reverse-engineering, debugging, and evaluating embedded network programs. We also expect the system to serve as a powerful teaching tool.

5.4 Evaluation

The Visualization Toolkit and Visualization Testbed enable control-flow visualization of embedded network systems. In this section, we evaluate the tools in terms of their suitability of purpose, the rate at which trace events can be captured, and the resource overhead imposed by instrumenting target applications. Our evaluation of the implementation is based on the following objectives. The implementations must:

- **O3.1** Enable users to better understand the flow of control through event-based programs by providing visualizations of both static and dynamic control-flow information
- **O3.2** Enable users to control the detail of the visualizations by applying static and dynamic filters to the available data

- **O3.3** Be suitable for visualizing common nesC execution patterns

- **O3.4** Be suitable for use on resource-constrained devices

The following subsections detail our evaluation of the Visualization Toolkit and Visualization Testbed. Subsection 5.4.1 details the properties of the tools that make them suitable for visualizing the control-flow of nesC applications. Subsection 5.4.2 details the rate at which events can be captured. Subsection 5.4.3 details the resource overhead introduced by the instrumentation process.

### 5.4.1 Suitability of Purpose

The Visualization Toolkit and the Visualization Testbed enable users to visualize both static and dynamic control-flow information associated with nesC applications. The tools provide two visualizations: static system call graphs and dynamic UML sequence diagrams. These visualizations enable users to better understand the flow of control through event-based nesC programs (O3.1). Additionally, the tools allow users to filter events both statically and dynamically. Static event filters take the form of dialogs that enable users to select the functions of interest. Dynamic event filters present a similar interface, but also include the ability to isolate “trace windows”. These filters enable users to control the detail of the visualizations, allowing them to view only the information of interest (O3.2).

### 5.4.2 Capture Rate

The following paragraphs detail our analysis of capture rate provided by the event recording service for both the Visualization Toolkit and the Visualization Testbed.

**Visualization Toolkit.** The capture rate of the event recording service plays an important role in assessing the Visualization Toolkit’s utility. If the maximum capture rate is low, the tool will not be suitable for fine-grained visualization, or for systems in which
### Table 5.2: Visualization Toolkit Event Logging Time

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Buffer Count</th>
<th>Time (seconds)</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3,448</td>
<td>0.008</td>
<td>1.051</td>
<td>0.009</td>
<td>0.032</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1,856</td>
<td>0.011</td>
<td>1.054</td>
<td>0.014</td>
<td>0.045</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1,216</td>
<td>0.014</td>
<td>1.058</td>
<td>0.185</td>
<td>0.058</td>
<td></td>
</tr>
</tbody>
</table>

Instrumented actions will be executed with high frequency. To evaluate the capture rate of the visualization toolkit, we developed a test application to measure the recording time over a continuous stream of events. The application repeatedly logs a full event buffer (i.e., 70 events) and records the duration of each call. The process is repeated for three different event record types, ranging in size from 1 byte to 3 bytes. This is to account for the fact that event records are dynamically sized based on the instrumentation set to minimize storage requirements. The tests were performed using the Tmote Sky platform.

A summary of the experimental results is shown in Table 5.2. Each row corresponds to a single run of the test application using event records of the specified size. **Buffer Count** indicates the number of buffers written to log storage before the target volume reached capacity. The remaining columns have the obvious meanings. Despite the large variation in recording time witnessed during each run, the results are favorable. In the worst case, it took 1.058 seconds to log 70 records of 3 bytes each. Hence, given the dual-buffer implementation, the recording service can handle approximately 70 events per second, independent of the inter-arrival rate. (We note that the maximum capture rate can be tuned by increasing the number of event buffers, but at the expense of additional overhead.) This makes the toolkit especially well-suited to the visualization of **bursty** execution patterns, in which a node periodically wakes to perform a dense series of actions and then resumes its idle state. Fortunately this pattern is representative of most embedded network applications. This makes the Visualization Toolkit suitable for visualizing common nesC execution patterns (O3.3).

---

7Recall that the Event Recording Service is implemented as a thin wrapper over the LogStorageC component provided by TinyOS. Given a fixed event record size, assessing the maximum capture rate is equivalent to evaluating the performance of LogStorageC for the target hardware (or simulation) platform.

8The variation is likely due to the underlying buffering strategy implemented by LogStorageC.
Visualization Testbed. The maximum capture rate of the Trace Recording Service has a direct impact on the utility of the Visualization Testbed. If the maximum capture rate is low, the achievable instrumentation density would be reduced. Similarly, it would not be possible to capture high frequency events. The additional log state required to correlate send and receive events increased the log record size, invalidating the analysis presented above. To evaluate the new maximum capture rate, we applied the same experimental setup.

A summary of the results is shown in Table 5.3. In the worst case, it took 1.017 seconds to log a full event buffer (i.e., 50 records of 4 bytes each). Hence, in the worst case, the service can handle 50 events per second independent of the inter-arrival rate. Again, we conclude that this performance is acceptable for a wide range of testing scenarios. This makes the Visualization Testbed suitable for visualizing common nesC execution patterns (O3.3).

### 5.4.3 Resource Overhead

The resource overhead introduced during the instrumentation process is another important evaluation metric. If the overhead is high, the achievable instrumentation density will be low, resulting in low-fidelity visualizations. This would also limit the potential integration outlets for the toolkit. To evaluate the impact on resource usage, we applied the toolkit to each of the sample applications included as part of the TinyOS 2.0 distribution. The instrumented systems were used to evaluate the base and incremental impact of probe insertion on application memory usage. The expansion of the Visualization Toolkit and its integration with the NESTbed produced no additional resource overhead; therefore, the

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Buffer Count</th>
<th>Time (seconds)</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1.280</td>
<td>0.014</td>
<td>1.017</td>
<td>0.018</td>
<td>0.052</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: Visualization Testbed Event Logging Time
results presented here represent the resource overhead for both the Visualization Toolkit and the Visualization Testbed.

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

The resource requirements under each configuration are shown in Table 5.4. The Baseline columns correspond to the first configuration; the EEPROM Logging columns correspond to the second configuration; and the Full Instrumentation columns correspond to the third configuration. The RAM results are summarized in Figure 5.11a, and the ROM results are summarized in Figure 5.11b. The results are consistently favorable. On average, the instrumentation toolkit introduces a base cost of approximately 256 bytes of RAM and 6239 bytes of ROM. (Again, the size of the logging buffers can be tuned to reduce RAM overhead, but at the expense of reducing the maximum achievable capture rate.) Factoring out the overhead introduced by LogStorageC yields the additional cost of

<table>
<thead>
<tr>
<th>Application</th>
<th>Baseline RAM</th>
<th>Baseline ROM</th>
<th>EEPROM Logging RAM</th>
<th>EEPROM Logging ROM</th>
<th>Full Inst. RAM</th>
<th>Full Inst. ROM</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseStation</td>
<td>1,610</td>
<td>13,160</td>
<td>1,766</td>
<td>18,902</td>
<td>1,922</td>
<td>19,686</td>
</tr>
<tr>
<td>Blink</td>
<td>55</td>
<td>2,654</td>
<td>216</td>
<td>8,490</td>
<td>360</td>
<td>9,276</td>
</tr>
<tr>
<td>MViz</td>
<td>1,722</td>
<td>33,640</td>
<td>1,854</td>
<td>38,402</td>
<td>2,003</td>
<td>39,258</td>
</tr>
<tr>
<td>MultihopOscilloscope</td>
<td>3,368</td>
<td>28,624</td>
<td>3,498</td>
<td>33,410</td>
<td>3,650</td>
<td>34,284</td>
</tr>
<tr>
<td>MultihopOscilloscopeLqi</td>
<td>2,451</td>
<td>24,438</td>
<td>2,589</td>
<td>29,286</td>
<td>2,729</td>
<td>29,808</td>
</tr>
<tr>
<td>Null</td>
<td>4</td>
<td>1,418</td>
<td>184</td>
<td>8,258</td>
<td>330</td>
<td>9,038</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>372</td>
<td>15,320</td>
<td>504</td>
<td>20,166</td>
<td>652</td>
<td>21,024</td>
</tr>
<tr>
<td>Powerup</td>
<td>4</td>
<td>1,472</td>
<td>186</td>
<td>8,272</td>
<td>330</td>
<td>9,060</td>
</tr>
<tr>
<td>RadioCountToLeds</td>
<td>286</td>
<td>10,794</td>
<td>420</td>
<td>15,712</td>
<td>568</td>
<td>16,566</td>
</tr>
<tr>
<td>RadioSenseToLeds</td>
<td>344</td>
<td>15,158</td>
<td>474</td>
<td>20,004</td>
<td>620</td>
<td>20,888</td>
</tr>
<tr>
<td>Sense</td>
<td>95</td>
<td>7,184</td>
<td>256</td>
<td>12,762</td>
<td>400</td>
<td>13,600</td>
</tr>
</tbody>
</table>

Table 5.4: Base Overhead (RAM/ROM)
the recording service in applications that already include LogStorageC: 147 bytes of RAM and 802 bytes of ROM, on average. This overhead is suitable for use with applications that run on resource-constrained devices (O3.4).

It is also important to consider the incremental cost of each probe. In general, the instrumentation cost of a given action varies based on the number of exit paths it contains, as well as the compiler optimization context in which the probes appear. To give a sense of the typical cost associated with a single probe, we compiled several versions of the Blink application, increasing the number of probes from one version to the next. The first version
contained a single *entry* probe; the second contained an *entry* probe followed by an *exit* probe; the third added a second *entry* probe; etc. The underlying records used to store events were 1 byte each. The resource requirements are summarized in Figure 5.12. Again, the results are favorable. While there is some variation in incremental cost, a new probe requires approximately 12 additional bytes of ROM on average. There is no incremental RAM expense, assuming that an additional probe does not increase the minimum required size of the underlying event records. Again, the resource requirements are suitable for use with applications that run on resource-constrained devices (O3.4).

### 5.5 Research Contributions

We have presented the design and implementation of the Visualization Toolkit and the Visualization Testbed, implementations of a control-flow visualization framework for nesC applications — Contribution 3 of this dissertation. We identified design desiderata and showed how our implementations satisfy those desiderata. We detailed the process of applying the Visualization Toolkit to nesC applications to collect and visualize static and dynamic function trace data. We described how the Visualization Toolkit was expanded
and integrated with the NESTbed (described in Chapter 3) to produce the Visualization Testbed. We showed how in addition to the services provided by the Visualization Toolkit, the Visualization Testbed provides “linked” UML sequence diagrams that enable users to correlate message send and receive events across a network of devices. Using example scenarios, we showed how users can apply both these tools to collect and visualize static and dynamic control-flow information for nesC programs. The example scenarios also illustrate how static and dynamic filters can be used to visualize only the data that is of interest to the user. Finally, we evaluated the tools in terms of their suitability of purpose, the rate at which events can be captured, and the resource overhead introduced by the application of the tools.

Prior to this work, there were no techniques to visualize nesC control-flow. Unlike sequential programs where a sequence of statements is executed linearly, systems for the nesC platform are event-driven. As a result, as nesC programs became large, developers were unable to imagine all possible paths of control through the implementation components — the interleavings became unmanageable.

The Control-Flow Visualization Framework solves these problems. The framework enables users to visualize the static system call graph associated with a program, as well as collect dynamic trace data from a running program and visualize that data as a UML sequence diagram. These diagrams help developers to understand the control-flow through event-driven nesC applications. The framework also enables the collection of network send and receive events. This additional dynamic trace data is integrated with the UML sequence diagrams to provide a set of “linked” diagrams. These linked diagrams enable users to understand the causal relationships between distributed events more easily. The framework also provides users with the ability to filter the visualization data statically and dynamically. Static filters enable users to predetermine the set of events to collect. Dynamic filters enable users to visualize a subset of the collected data. These filters make it easier for users to identify and view only the events of interest. This contribution has, and will continue to improve the development of embedded network systems.
Chapter 6

Related Work

In this section we summarize related research in the areas of (1) embedded network system testing, debugging, and profiling, (2) program analysis and instrumentation, and (3) program visualization. Section 6.1 describes existing work in testing, debugging, and profiling; Section 6.2 considers existing work in program analysis and instrumentation; and Section 6.3 describes existing work in fault localization and program visualization.

6.1 Program Testing, Debugging, and Profiling

The difficulty of testing, debugging, and profiling embedded network systems is well-recognized. A number of tool-based solutions have been proposed to address these difficulties. We survey some of the most relevant here.

6.1.1 Network Simulators

Several hardware platform-independent embedded network simulators have been discussed in the literature [4,71,119]; hardware platform-specific embedded network simulators have also been described [65,92,99]. These tools have proven effective in providing initial measures of correctness and performance. The point of departure for our work, however, was the observation that they have not supplanted the need for physical experimentation.
Simulators offer limited fidelity with respect to modeling wireless signal propagation and interference \cite{98, 120}, as well as in capturing the behavioral subtleties of underlying hardware platforms. Consequently, our focus has been on infrastructure support for physical experimentation, debugging, and profiling (Chapter 3).

Hybrid Simulators. Hybrid approaches that combine aspects of physical experimentation and network simulation have also been proposed. The key idea is to identify system aspects that cannot be faithfully simulated and factor them out to the physical world. In the context of embedded network simulation, this typically involves deferring communication to physical devices while simulating other system aspects (e.g., application execution, network traffic). Hybrid testbeds have been used to evaluate both wired and wireless ethernet networks \cite{32, 116, 121}. Similar approaches have been used in the context of embedded network systems. The EmStar development platform \cite{45}, for example, targets microservers, Linux-based embedded nodes with computational resources equivalent to a PDA device. The platform can also target Linux-based desktops and servers. Applications developed using EmStar can be simulated using EmSim, a simulator that allows physical radios to be used in place of simulated network channels. EmTOS \cite{46}, an extension of EmStar, allows applications developed using nesC and TinyOS to be simulated by a microserver (or desktop/server). As a result, the architecture supports hybrid simulation, enabling designers to experiment with alternative realizations of physical network interfaces. The SeNeTs framework \cite{6} provides similar features, but offers support for large-scale simulation through the use of distributed processing. While hybrid frameworks have yielded important research results, they have not addressed the fidelity issues associated with simulating mote hardware. They cannot, for example, be used to gather precise results concerning the effects of hardware interrupts, load-induced execution anomalies, or other complex phenomenon that cannot be faithfully simulated. Moreover, results obtained using a particular network interface and supporting software stack rarely apply to other interfaces and network drivers. By contrast, our work supports pure physical experimentation using standard hardware components, offering the highest degree of experimental fidelity.
Network Testbeds. We are not the first to describe a network testbed designed to support pure physical experimentation. Several 802.11 efforts have been discussed in the literature [13,58,81,87], and more recently, testbeds focused on embedded network systems have emerged. Our work aligns most closely with the latter category. We consider some of the most important testbed efforts representative of the current state-of-the-art.

Harvard’s MoteLab testbed [112,114] was one of the first embedded network testbeds discussed in the literature. The physical network includes 190 Tmote Sky [78] devices. Each mote is attached to an ethernet-based gateway device [77], allowing the network to be reprogrammed from a centralized server. The server exposes a web interface that allows users to upload executable application images and to configure the deployment of those images on the physical network. The system also allows users to upload Java classes that can be used to log USB data. The NESTbed approach to creating network gateways for injecting packets from a remote location is based on a similar feature available in MoteLab. In contrast to the NESTbed system, however, MoteLab is batch-based rather than interactive; submitted jobs are queued for later execution. As a result, the system does not support real-time source- or network-level profiling\(^1\), nor does it support the injection of transient state faults. The design is also image-centric, requiring users to generate application images, as well as to construct the Java classes that parse application data transmitted over the USB port. In addition to the productivity benefits provided by the NESTbed system, its source-centric design introduces opportunities for automated source-level analysis and instrumentation. Finally, the MoteLab server appears to be closed; it does not seem to expose an API for programmatic control, restricting users to a single web interface for all experimentation tasks. There is no equivalent, for instance, of the NESTShell scripting interface, nor an apparent mechanism to add such an interface.

More recently, Ohio State deployed the Kansei testbed [2,38,82]. The supporting physical network is one of the largest to date, with over 400 devices. The testbed supports experimentation over multi-tiered networks; the deployment includes Extreme Scale [35],

\(^1\)This discounts the possibility of forwarding raw packet data through a gateway for remote inspection.
Tmote Sky [78], and Trio motes [36]. The basic hardware architecture is similar to that of MoteLab. Motes are attached to ethernet-based gateway devices [23] and are programmed through a centralized server. A key point of novelty in Kansei is its focus on sensing experiments. Sensor nodes are housed in stationary, portable, and mobile arrays. Portable and mobile arrays are used to collect field data for ex post facto analysis. Alternatively, the arrays can be used to inject data into the (larger) stationary deployment using a forwarding system similar to that of MoteLab and NESTbed. Unlike these systems, however, Kansei includes support for sensor stream scaling. Perhaps most interesting is the third option which generates large-scale sensor streams by replaying pre-recorded data (usually from a smaller portable array) with temporal and spatial shifts. The software architecture also provides some support for job coordination.

Kansei is well-suited to batch-style experimentation, especially when the experiments are focused on high-fidelity sensing of parameters that cannot be captured in a laboratory context. The NESTbed system offers a complementary design; it is engineered to support interactive use, with a focus on software experimentation (i.e., testing and debugging the software at the time of its development). Like MoteLab, Kansei does not provide real-time profiling or fault injection support. It also provides limited support for logging network traffic [83]. Further, Kansei is image-centric, precluding source-level analysis and instrumentation. One consequence seems to be that developers are required to integrate specialized Kansei components before compiling and uploading their application images. Finally, the degree of controllability provided to external applications by the Kansei API is unclear. The PHP-based design seems to suggest a closed system, precluding the addition of interface extensions such as the NESTShell scripting interface provided by the NESTbed system.

MoteLab and Kansei exemplify testbed development projects underway at research institutions around the world (e.g., [16,17,49,56,105,107,111])\(^2\). While these testbeds share

\(^2\)It may be useful to note that MoteLab, Kansei, the Deployment Support Network, and other testbeds include integrated health monitoring services. The basic approach is to poll each device periodically to determine whether it is in a programmable state. Unresponsive nodes are avoided by manual and automatic
similarities with our work, they are principally batch-based, image-centric, and closed. By contrast, the NESTbed design is interactive, source-centric, and open.

6.1.2 Other Tools

In addition to testbed infrastructures, other related development and testing tools have also recently been described in the literature. Most relevant to our work are tools designed to improve runtime observability and controllability of network software. The Deployment Support Network (DSN) [37] is a key example. The approach is to connect a secondary device, the “DSN node”, to each target mote. The DSN nodes form a reliable out-of-band backbone for controlling and observing a target mote network. The current implementation uses Bluetooth-based BTnodes [39] for the out-of-band backbone. These devices are managed through a centralized server that exposes an RPC interface to remote processes. The interface is used to deploy new application images, transmit and receive messages from target devices and monitor the status of DSN nodes and their targets. In-and out-of-network buffering strategies are used to provide reliable communication. The key benefit of the toolkit is to eliminate the need for wired mote connections. In effect, the toolkit serves as a replacement for USB connections in Tmote-based (and other sensor) deployments. It may be worth noting that the DSN approach has been used to construct a heterogeneous network testbed consisting of 66 nodes at ETH Zurich [40]. The testbed has features (and consequent limitations) analogous to those of the MoteLab testbed.

Nucleus [100] is a lightweight query system for TinyOS that exposes nesC variables as attributes. Exposed attributes can be read and written at runtime using the Nucleus Java Library. Marionette [117] is a significant extension of Nucleus to support RPC-based interactive development and debugging. The system provides a Python interface for exploring static program structures (e.g., modules, type declarations), reading and modifying program state at runtime, and invoking nesC commands. Like the NESTShell interface, Marionette allocation strategies. Because the NESTbed system is intended for interactive use, users are notified of device problems at the point of installation (as indicated by programming failures). Unresponsive nodes can be power-cycled through the NESTbed interface. Hence, while useful in batch-based systems, the benefit of periodic health monitoring is unclear in the context of the NESTbed design.
enables developers to script debugging and profiling activities. It is not, however, tailored for testbed experimentation; it lacks services for managing projects and deployment configurations, reprogramming devices, constructing network gateways, and others. Further, mote interactions are handled in-band, limiting throughput and reliability. Marionette’s integration with a popular object-based scripting language, however, is a point of advantage over the NESTShell interface.

Finally, it may be worth noting that the NESTbed graphical interface bears some similarity to existing integrated development environments for nesC and TinyOS. In particular, it shares design characteristics with the various Eclipse plugins for TinyOS [89,90,103], as well as TOSDev [73, 74]. By contrast to the NESTbed system, however, these tools are focused on providing syntactic assistance (e.g., syntax highlighting, code completion) and managing source distributions and component dependencies. They provide support for programming a single device; they do not provide testbed-related features.

6.2 Program Analysis and Instrumentation

The difficulties associated with developing flexible and efficient analysis and instrumentation libraries are well-recognized. In the domain of imperative programming languages, a number of solutions have been proposed to reduce these difficulties. The solutions target object-, intermediate-, and source-level program representations. Recall that these solutions are inapplicable to nesC because of its unique features, including language-level synchronous and asynchronous events, language-level tasks, component wirings, and fanning of function calls and returns. In short, the programming model is fundamentally new. We have, therefore, focused our attention on supporting program analysis and instrumentation for the nesC platform (Chapter 4).
6.2.1 Object-Level

Object-level program analysis and instrumentation tools target the compiler-generated binary program images. This post-compilation process is independent of the source language and compiler. (In fact, the original source materials are not needed.) Many object-level tools have been presented in the literature for various hardware and software platforms. Tools for binary executables include ATOM [94], EEL [64], PatchWrz [12], Etch [88], Dyninst [10], Pin [67], LOPI [57], and others. Tools for Java bytecode include BCEL [24], JOIE [18], SERP [115], and SOOT [108]. Here we focus our discussion on two contributions that represent the state-of-the-art in object-level analysis and instrumentation for both binary executables and Java bytecode, namely Valgrind and ASM.

Valgrind [80] is a source language-independent binary instrumentation framework for Linux. It consists of a command-line tool that accepts application binaries to be executed, disassembles those binaries into an intermediate representation (IR), instruments the IR with analysis code, and converts the IR back into machine code that is then executed. The framework supports the construction of a wide range of specialized analysis and instrumentation tools, the most popular of which is Memcheck [91]. Memcheck enables users to detect a range of memory errors, including multiple frees of dynamically allocated memory and memory leaks. Although Valgrind is a useful framework in the domain of desktop-and enterprise-class applications, it cannot be applied to the domain of embedded network systems. Its on-the-fly instrumentation approach makes it too heavyweight for embedded network systems. Our approach, however, instruments the application source before it is compiled and installed. It introduces a small ROM overhead to store the instrumented code and a small runtime overhead to execute that code. Also, Valgrind supports only the Linux operating system on Intel and PowerPC architectures, making it inapplicable to TinyOS applications running on the microcontrollers found in embedded network devices. Our source-based approach supports TinyOS and is independent of the underlying hardware architecture. Finally, Valgrind is source language independent; its API exposes a custom IR developed for the tool. Our API, however, exposes the underlying program as a set of
ASTs containing nodes that correspond to elements of the nesC language, with which the
developer is already familiar.

ASM [9] is a framework than enables users to manipulate Java bytecode. ASM
parses the binary class files generated by a Java compiler, restructures the bytecode and
either writes the restructured bytecode to disk or executes it on-the-fly. It provides two
APIs enabling developers to interact with the systems under analysis or instrumentation.
The first is an \textit{event-based} API. With the event-based API, as ASM reads class files, it fires
events associated with the elements of the class (e.g., fields, method declarations). This
behavior is similar to that exhibited by SAX-based XML parsers. While the event-based
approach is efficient for analysis activities, it does not easily support instrumentation. The
second API is \textit{tree-based}. This API exposes the Java class files as ASTs and provides a set
of visitors for traversing and modifying those trees. This is similar to the document object
model (DOM) XML representation. Unlike our approach where the nodes of the ASTs
represent high-level language constructs, the nodes within ASM ASTs represent bytecode
instructions, requiring developers to be familiar with the low-level bytecode. Also unlike
our approach, where the API provides methods for simplifying AST traversal, modification,
and generation, all such activities are handled using only visitor-based approaches. Finally,
although bytecode manipulation frameworks, such as ASM, are useful for devices designed
to execute Java bytecode directly; most embedded network systems are currently designed
for the nesC/TinyOS platform.

\subsection*{6.2.2 Intermediate-Level}

Some programming languages, such as C++, are notoriously difficult to parse cor-
rectly [41]. To ameliorate this difficulty, an approach to processing the compiler-generated
IR of C++ applications has been developed. Unlike C++ source, the IR is simple to parse.
The $g^{tre}$ tool chain [61] is a reverse-engineering tool for C++ that targets GENERIC, one
such IR used by the $g$++ compiler, to provide reverse-engineering and program analysis
services. These services are exposed through an API that enable users to access the ab-
abstract semantic graphs (ASGs) representing individual compilation units. The API enables users to iterate over the elements of the graph, or, like ASM, to access the elements using a visitor-based approach. The g^4re API, like our approach, provides program-level information in terms of the source language. However, unlike our approach, which provides expression-level access to program information, g^4re provides information down to only the declaration level; expressions are not included. Also, g^4re supports only analysis tasks — it is not capable of performing any type of program instrumentation or source regeneration. Finally, while useful for analyzing C++ applications, g^4re cannot be applied to nesC programs.

6.2.3 Source-Level

Source-level program analysis and instrumentation tools target application source code. Analysis tasks can be achieved either by instrumenting an existing compiler to collect the desired information, or by developing a custom parser for the language under analysis. Instrumentation tasks can be achieved by enabling these approaches to update and regenerate the target application’s source code. The gccXfront [51] approach, for example, uses a modified version of the gcc parser to generate XML files that represent the structure of C, C++, and Java programs under analysis. The generated XML files are used as input to a Java-based tool that enables users to view the generated XML, graphically navigate the tree-like structure of the generated XML, view the XSLT stylesheet used to transform the XML into a more (human) readable form, and view processed XML documents after the XSLT transformations are applied. Unlike our approach, which provides an API enabling users to analyze a target application, gccXfront produces XML files that contain program information. Other tools are necessary to read those XML files and perform the analysis activities. Also, unlike our approach, gccXfront provides no means of performing program instrumentation. While gccXfront is useful for analyzing C, C++, and Java programs, it cannot be applied to nesC applications.
Tools supporting aspect-oriented programming (AOP), such as *AspectJ* [59] and *AspectC++* [93], can be considered special cases of source-level instrumentation tools. With AOP, concerns that cross-cut modules are factored into modular *aspects*. The aspects include code segments that are to be *woven* into a collection of modules and a specification of where the code fragments should be woven. AOP tools have been used to instrument programs with debugging and monitoring code [68], as well as to instrument programs with code to check temporal invariants [44]. Unlike our approach, which enables users to instrument the system at any point in the code, AOP techniques can only be applied at well-defined *join points*. Also unlike our approach, which enables users to both analyze and instrument their applications, AOP can be used only for instrumentation.

More closely related to our approach are tools that implement parsers for the language under analysis. One such tool is *Columbus* [41], a reverse-engineering tool for C++. The tool includes support for parsing a set of C++ source files, linking those files (i.e., resolving interdependencies among components in different source files), filtering unwanted details and exporting the filtered data to output files. The filtered data can then be processed by other tools to perform program analysis and visualization. Columbus, however, enables only program analysis; program instrumentation is not possible. Unlike Columbus, which targets C++, our approach targets the nesC programming language.

None of the previously described work enables analysis and instrumentation activities for nesC-like languages. The analysis features of our toolkit enable users to determine the type of each function (e.g., task, command, event). Each of these function classes exhibit different calling scopes and behaviors within the context of an application. Unlike the languages supported by existing tools, nesC components are completely decoupled. Our approach enables users to analyze the wirings used to associate used interfaces with the implementations that provide the functionality, as well as to update these wirings to add additional realizations. Finally, unlike the languages supported by existing tools, function calls and returns can *fan*. A single function call in nesC source can correspond to the execution of multiple function bodies, depending on the number of components that are wired
to the calling interface. Our API enables developers to extract that fanning information — to be able to determine statically the set of functions that will be executed when a call is made.

6.3 Fault Localization and Program Visualization

That embedded network systems are difficult to construct and debug is hardly a new observation. The exploration of techniques and tools for reducing this difficulty continues to be a major research thrust within the community. Here we survey some of the most significant fault localization efforts reported in the literature. We additionally survey key results in both local and distributed program visualization. While these topics have received relatively little attention in the domain of embedded network systems, they have a long and rich history in the domain of desktop systems. Our focus has been on visualizing the runtime behavior of embedded network systems (Chapter 5).

6.3.1 Fault Localization

Several authors have described techniques for detecting and localizing faults in embedded networks. Ramanathan et al. [86] present an approach based on comparisons between actual and expected network traffic patterns. While helpful in identifying node-level fault candidates, the approach does not aid in identifying source-level problems. In contrast, Krunic et al. [62] focus on providing source-level assistance. The authors describe a diagnostic system designed to trap program faults before they can disable the hosting device. The system includes a network interface for collecting context information related to a fault, including runtime trace information. Our approach to encoding trace information using numeric tokens is similar to the approach discussed in [62].

Improving runtime visibility has been another important focus in the literature. Tolle and Culler [101] describe a network management system that enables developers to expose attributes as part of a program implementation. Attributes are encoded manually
and can be read and written across a network at runtime from a basestation. The system additionally provides the ability to log events at manual instrumentation points and to exfiltrate log data for later analysis. Again, the event encoding approach is similar to ours. Whitehouse et al. [117] extend this work; they describe a system for accessing program state without the need for manual attribute encoding. The toolkit additionally provides remote procedure call capabilities from a basestation to a network node. While these systems have proven useful in improving runtime visibility, they provide little insight into the path sets underlying a system, or the particular paths chosen during a run — the focus of our work.

### 6.3.2 Program Visualization

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

Orso et al. [85] describe an approach to visualizing program execution data collected from deployed software. Their focus is on developing visual abstractions that scale to large datasets and that can be tailored for use across different types of execution data. The approach includes three visual abstractions used to represent statement-, file-, and system-level views of a collected dataset, respectively. The authors also describe approaches to filtering and summarizing runtime data across multiple runs. While the utility of the authors’ work has been vetted in a number of contexts, none of the three visual abstractions are well-suited to reasoning about program control flow. Our work, on the other hand, is specialized for this purpose.

Closer to our work is that of Lange and Nakamura [63]. The authors describe a program visualization toolkit for C++ applications that combines static structural information and dynamic trace information to generate object-centric views of program behavior. The toolkit components are similar to ours; they include an instrumentation system, an execution trace recorder, and a program database from which static program information can be retrieved. The generated visual representations include object creation and lifetime graphs, as well as object-centric call graphs. A simple selection interface enables filtering
on classes and methods. Malloy and Power [70] describe a similar, but more advanced visualization system for C++ applications. The generated visual representations include class and method call graphs, UML communication diagrams, and UML sequence diagrams. The system additionally includes static and dynamic event filtering and supports dynamic visualization during the execution of a program under test. In contrast to the work of these authors, our work targets TinyOS applications. The programming model (and associated language) is fundamentally different than that provided by C++. Moreover, our approach operates in an asynchronous execution environment under tight resource constraints.

Finally, it is worth noting that a number of groups have recently released development environments for TinyOS [74, 89, 90]. The environments include a subset of the features found in standard development environments, including syntax highlighting, automated code completion, compilation support, etc. The environments additionally provide support for visualizing the static structure of TinyOS applications. The representations range from simple hierarchical component views to more detailed representations of component bindings and call graph structure. The tools do not, however, support visualization of dynamic program behavior, which is our main focus.

6.3.3 Distributed Program Visualization

Reducing the difficulty of embedded network system development continues to be a major research thrust within the community. Recent efforts to address this problem propose a range of solutions. Some authors have described new programming models and abstractions to simplify development [15, 72]. Others have focused on frameworks for improving runtime state visibility [101, 117] and diagnostic tools for capturing execution data to support fault localization [62, 86]. Surprisingly, visualization techniques for improving program comprehension have received little attention. Indeed, our work appears to be the first to consider program visualization in the context of any embedded network platform. We view this work as an important first step in a much broader, community-based initiative.
In the desktop space, our focus is not unique; other authors have described visualization systems tailored to reasoning about distributed behavior. Moe and Carr [76], for example, describe an approach to collecting trace data from CORBA-based systems using *interceptor* components. Scatter plots are used to summarize the collected data and to identify undesirable behaviors. A plot comparing successful and unsuccessful remote method invocations, for instance, can reveal repetitive network faults. Wu et al. [118] describe a trace collection and visualization framework for SMP clusters. The primary visualization mechanism is a *space-time* plot, used to summarize logged trace data across processors, threads, and event categories. A key contribution of their work is a clock synchronization technique used to ensure timestamp consistency across distributed events. They also include message instrumentation to correlate transmission and reception events, as in our approach. Topol et al. [102] describe a more sophisticated toolkit for visualizing the behavior of cluster-based systems. The toolkit provides a broad range of real-time views for summarizing system load, message traffic, memory utilization, and other performance characteristics. Additional views are provided for postmortem analysis, including one tailored to visualizing the message exchanges between nodes. Beyond the significant platform differences already emphasized, our work differs in its emphasis on *source-level* program understanding. The generated graphical views are consequently much different.

Closest to our work is that of Briand et al. [8]. The authors describe a trace-based approach to reverse-engineering UML sequence diagrams for distributed systems constructed using *Java RMI*. Each diagram captures a *local* projection of *global* behavior. Distributed interactions are captured using diagrams that combine objects across hosting platforms; object names are prefixed with numeric identifiers to indicate their hosts. While suitable to systems distributed over a small number of devices, local projections are unwieldy for large systems. Indeed, with only a handful of devices, it can be difficult to understand the relationships between local behaviors and global behaviors. Hence, in our approach, we provide *linked, local* views of distributed behavior. Local views are navigated by travers-
ing corresponding send and receive events. Further, our system is integrated as part of an embedded network testbed to provide a unified environment for deploying, debugging, profiling, and visualizing network applications.
Chapter 7

Conclusion

Embedded network systems promise to bring the vision of ubiquitous computing to fruition. The lowest tiers of these systems are composed of “motes”, inexpensive devices that sense, process, and communicate environmental stimuli to other motes or basestations. Applications for these systems are most often developed using the nesC programming language and the TinyOS operating system. The platform has become the de facto standard for embedded network system development and has thus been the focus of this dissertation.

Unfortunately, the applications that execute on these networks are notoriously difficult to design, build, and debug. The observation underlying this dissertation is that these difficulties are due in large part to the lack of an integrated, general-purpose framework for analyzing, instrumenting, and visualizing code written for the nesC platform. We have identified three sub-issues associated with this problem. First, there are no suitable interactive frameworks for analyzing the runtime behavior of embedded network applications (e.g., testing, debugging, profiling). The lack of such a framework makes debugging and profiling applications tedious and time-consuming. Second, there are no suitable general-purpose frameworks for performing static analysis and instrumentation activities on nesC applications. The lack of such a framework limits the software engineering tools and techniques that can target the nesC language and programming model. Finally, there are no techniques that enable developers to trace the flow of control through nesC
applications. Without such techniques and tools, understanding the dynamic behavior of 
nesC programs is difficult. Section 7.1 summarizes our contributions, contributions which 
address these problems. Section 7.2 concludes with the expected impact of our work on the 
state of embedded network system development.

7.1 Contribution Summary

We have described the following contributions:

**Contribution 1 – Interactive Testing Framework.** Motes provide few mecha-
nisms to expose their runtime behavior. Debugging and profiling messages sent to a bases-
tation can provide more information; however, the instrumentation logic must be integrated 
a priori. Prior to our work, the “interesting” portions of an application needed to be iden-
tified and manually instrumented in advance. As a result, developers did not benefit from 
the techniques used to improve programmer productivity in other development domains.

We described an Interactive Testing Framework for embedded network system de-
velopment that addresses this issue. First, the approach is source-centric and enables auto-
mated analysis and instrumentation of nesC applications. Analysis activities reveal message 
structures used by an application and enable developers to monitor and record instances 
of those messages received from motes in real time. Analysis activities also enable nesC 
module variables to be identified and later profiled. Instrumentation enables new software 
components to be incorporated into an application prior to being compiled and installed, 
allowing the testing framework to, for example, perform component substitution without 
developer intervention. Second, the approach enables users to query the values of module 
variables across a deployment, providing users with a “window” into the state of a running 
system through which they can analyze their applications. Third, the approach enables 
users to modify variable values at runtime, enabling users to inject transient faults into 
their systems. Finally, the approach is server-centric and open. The services provided by 
the system are implemented by a set of server-side Java RMI objects. Thin clients leverage
those APIs to provide services to users. Two client implementations are provided: a graphical application and a command-line application. The graphical application provides an easy-to-use drag-and-drop interface that is well-suited for novice users. The command-line application provides UNIX-like scripting services that enable both interactive and batch-based experiment automation.

We evaluated our implementation of the framework in terms of four objectives. The implementation was required to: (1) aid in debugging embedded network systems, (2) aid in testing embedded network systems, (3) enable performance analyses of embedded network systems, and (4) enable runtime profiling of embedded network systems. We showed that the NESTbed implementation meets each of these objectives.

**Contribution 2 – Analysis and Instrumentation Framework.** Prior to our work, there were no suitable techniques for performing source-level analysis and instrumentation in the context of the nesC programming language; these tasks were performed manually by developers. The process of performing analysis and instrumentation was tedious, time-consuming, and error-prone.

We described an Analysis and Instrumentation Framework for nesC development that addresses these issues. The approach includes analysis and instrumentation techniques that accommodate the novel features of nesC. These features include one-to-one and one-to-many component wirings, asynchronous events, tasks, and many others. To enable these techniques, the framework provides an API that allows users to develop new software engineering tools easily, as well as a visualization tool that aids users in comprehending the ASTs corresponding to their programs – ASTs on which the API acts. The API consists of three types of methods: (1) traversal, (2) modification, and (3) generation. Traversal methods simplify the process of navigating the ASTs, enabling users to identify portions of programs for analysis or instrumentation quickly. The methods enable users to follow the AST’s structure, allowing them to, for instance, identify the functions defined within a module, and the function invocations made within each function. Modification methods simplify the process of making changes to the ASTs, enabling users to instrument exist-
ing source code. The most interesting of these methods enables users to eliminate generic components from a nesC source base. Similar to eliminating template classes from a C++ program, the ability to instantiate generic components provides users with an analysis and instrumentation model that matches their conceptual model of the programming environment. Generation methods simplify the process of generating new code segments – segments that can be inserted into existing programs using modification methods. These methods can, with a single method call, construct AST fragments that consist of dozens of objects – objects users would have to create and compose manually without the API. Each of these types of methods are intended to simplify the most common analysis and instrumentation tasks.

In addition to the API, the framework also provides an AST visualization tool. This tool is used in concert with the API methods described above to help users to better understand the structure of the ASTs being traversed, modified, or generated. The development model is cyclic. Users can, for example, use the AST visualizer to understand the organization of an AST. Using that information, they can apply the traversal API to navigate to a point of interest within a program. Next, they can use the visualization tool to help them understand how the AST represents the source at that point in the program. Using this information, they can, for instance, generate a new program segment and inject it into the existing system.

We evaluated our implementation of the framework in terms of three objectives. The implementation was required to: (1) enable hardware platform-independent software engineering tools that target the nesC platform to be developed quickly and easily, (2) be suitable for the development of both interactive and batch-based software engineering tools, and (3) enable analysis and instrumentation tasks to be performed on desktop-class hardware. We showed that the nAIT implementation meets each of these objectives.

**Contribution 3 – Control-Flow Visualization Framework.** As nesC modules become large and involve many events, developers must try to imagine all potential paths of control through an implementation to understand its behavior. Systems written in nesC are
**event-driven**: statement sequences are not always executed linearly. Programs are comprised of a set of events that fire in response to external stimuli. Prior to our work, the possible interleavings of event handlers that could occur quickly became unmanageable.

We described a control-flow visualization framework for nesC that addresses this issue. The framework provides the ability to record the flow of control during an application run and to later visualize the information in a way that helps developers to understand the behavior of the application. First, the approach uses Contribution 2 to enable the automated insertion of control-flow recording code into nesC applications. Users apply static event filters to select only the code of interest for instrumentation. The injected code records runtime call sequence information to the EEPROM of the hosting device, allowing information to be collected from motes not attached to a basestation. Second, the approach enables users to download the recorded information to a basestation for visualization, helping developers understand the execution ordering in particular system runs. The visualizations take the form of annotated UML sequence diagrams. Colors are used to distinguish the context of each function call, as well as the function’s type (e.g., command, event, task). The visualization enables developers to identify unexpected control paths which could lead to incorrect program behavior.

We adapted this visualization technique to provide network-wide control-flow visualization services using Contribution 1. In addition to recording the sequence of functions that occur during a run, the following features are required. First, a unique sequence number is associated with each message sent from a mote. This sequence number is used to associate the transmission of a message on a node with the receipt of the message on other nodes. Second, *send* and *receive* events are recorded to EEPROM, along with the associated sequence number. The additional information enables control-flow on different devices to be properly correlated. This distributed information is then collected and used to provide a network-wide visualization of application control-flow. Again the visualizations take the form of UML sequence diagrams, one for each node of interest. These views are “linked” by corresponding send and receive events. This linked view enables users to navigate between
causally-related events in the system. As before, static filters are used to identify “interesting” portions of code. Additionally, dynamic filters are used to select the motes of interest and the trace “window” that is displayed. The approach enables developers to more easily understand the distributed behavior of their applications.

We evaluated our implementation of the framework in terms of four objectives. The implementation was required to: (1) enable users to better understand the flow of control through event-based programs by providing visualizations of both static and dynamic control-flow information, (2) enable users to control the detail of the visualizations by applying static and dynamic filters to the available data, (3) be suitable for visualizing common nesC execution patterns, and (4) be suitable for use on resource-constrained devices. We showed that the toolkit implementation meets each of these objectives.

7.2 Expected Impact

We have described an integrated, general-purpose framework for analyzing, instrumenting, and visualizing code written for the nesC platform. We believe that these contributions have had a significant and immediate impact on the state of embedded network system development. First, our Interactive Testing Framework complements existing testing frameworks and enables developers to debug their applications more quickly and easily. Developers are able to focus their attention on accomplishing their core objectives, rather than on instrumenting their source code with debugging calls. Second, our Analysis and Instrumentation Framework enables a wide-range of software engineering tools and techniques to be developed which target the novel features of the nesC language. The lack of such tools and techniques has, until now, limited their development. Finally, our Control-Flow Visualization Framework helps both novice and experienced nesC developers to understand the flow of control through their applications and to identify and remove unwanted paths of control. In short, our contributions has and will continue to improve the development of embedded network systems.
Bibliography


[42] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.


