nAIT: A Source Analysis and Instrumentation Framework for nesC

Andrew R. Dalton

Department of Mathematics and Computer Science, Western Carolina University
Cullowhee, North Carolina, 28723 USA

and

Jason O. Hallstrom

School of Computing, Clemson University
Clemson, South Carolina, 29634 USA

Abstract

Automated software engineering methods support the construction, maintenance, and analysis of both new and legacy systems. Their application is commonplace in desktop- and enterprise-class systems due to the productivity and reliability benefits they afford. The contribution of this article is to present an applied foundation for extending the use of such methods to the flourishing domain of wireless sensor networks. The objective is to enable developers to construct tools that aid in understanding both the static and dynamic properties of reactive, event-based systems. We present a static analysis and instrumentation toolkit for the nesC language, the de facto standard for sensor network development. We highlight the novel aspects of the toolkit, analyze its performance, and provide representative case-studies that illustrate its use.

Key words: Wireless sensor networks, nesC, static analysis, instrumentation

1 Introduction

For desktop- and enterprise-class system developers, automated software engineering methods have become important elements of the standard develop-
opment arsenal. Automation tools increase developer productivity, improve the reliability of new systems, and safeguard the reliability of legacy systems undergoing maintenance and evolution. The focus of this article is restricted to automated analysis and instrumentation methods. Even limited to this subset, however, automation benefits span a disparate range of tasks. Tools assist developers in reasoning about reference relationships [31, 41, 45], extracting diagrammatic source representations [18, 50, 51], visualizing runtime behavior [22, 47, 53], optimizing performance [23, 39], and understanding the behavior of malicious code [54]. Indeed, the benefits are numerous, extending far beyond this short list.

To facilitate the construction of such tools, a number of supporting library packages have been developed [3, 32, 57]. These packages provide common analysis and instrumentation services and expose those services in a manner that simplifies and expedites the construction of other tools. A tool developer might, for instance, use such a package to retrieve the abstract semantic graph corresponding to a given program. This structure might then be used as the basis for implementing points-to analyses that identify the memory locations that may be affected by the execution of a given program statement. (Side-effect analysis tools of this type are especially valuable during system maintenance and evolution.) Program comprehension benefits could then be further amplified: Instrumentation services provided by the supporting library could be used to inject logging probes at key system execution points. The logged data could be used to reverse-engineer UML sequence diagrams, correlate state modifications, or visualize control flow. The key point is that a wide range of tasks can be simplified by using a library package that provides general-purpose analysis and instrumentation services — a well-recognized fact.

The point of departure for our work is the observation that while a number of analysis and instrumentation toolkits exist for languages like C, C++, and Java, there are no analogous efforts for languages tailored for sensor network development. The principal contribution of our work is to address this significant gap. Our focus is on the nesC language [15, 30], the emerging standard for sensor network development.

But why concentrate on sensor networks? The focus is motivated by the burgeoning academic and commercial interest in these systems. Each is composed of tiny computing devices — ranging from the size of a matchbox to the size of a quarter — that sense, process, and communicate environmental stimuli. When deployed in large numbers, they serve as key enablers of the ubiquitous computing vision, creating a transparent compute fabric that can be installed across diverse environments. These systems are already transforming the way we coordinate with our peers [2, 28], study the ecology of our planet [20, 36, 48, 55], monitor our civil infrastructure [8, 17], and protect our
communities [1,43,49]. In short, sensor networks have the potential to have a profound impact on human experience.

Despite the rapid evolution of sensor network language and runtime platforms, few software engineering tools are available to help developers build, maintain, validate, and comprehend program implementations. The absence of such tools is likely to be an important factor contributing to the notorious difficulty of constructing and maintaining these systems, and the unexpected behaviors that are often exhibited in deployed networks. To enable the development of such tools and to enhance programmer productivity and system reliability, we present a general-purpose analysis and instrumentation toolkit designed for the nesC language. The intent is to provide a catalyst for extending the benefits of automated software engineering to this exciting new domain.

Before proceeding, it is natural to question whether a custom toolkit is required. Is it not possible to adapt an existing set of tools? Our answer to this question is no. The nesC language is fundamentally different from existing imperative languages due to the resource limitations of target hardware platforms and the unique requirements of sensor network applications. The analysis and instrumentation techniques embodied in existing toolkits simply do not apply. We return to this point in Section 2, where we highlight the main features of nesC by contrast to more traditional languages.

Contributions. The contributions of this article are as follows. First, we present the design and implementation of an analysis and instrumentation toolkit for nesC, the defacto standard programming language used in sensor network development. This is the first such toolkit to be described for any sensor network programming platform. Second, we present two case studies that demonstrate the utility of the toolkit in performing common analysis and instrumentation tasks. The first involves the construction of a debugging tool that provides services for monitoring state predicates that span module boundaries. The second involves the construction of a reverse-engineering system for producing UML sequence diagrams corresponding to particular program runs. Finally, we present a detailed analysis of the toolkit’s runtime performance characteristics and memory usage.

Paper Organization. In Section 2, we present a brief overview of the nesC language, with an emphasis on the points of novelty that preclude the application of existing analysis and instrumentation techniques. In Section 3, we present the design and implementation of the new toolkit. Section 4 presents the toolkit case-studies. A detailed quantitative analysis is presented in Section 5. Section 6 summarizes some of the most important elements of related work. We conclude in Section 7 with a summary of contributions and pointers to future work.
nesC (for “Network Embedded Systems C”) [15, 30] is a component-based dialect of the C programming language tailored for developing embedded network systems. While the application possibilities are numerous, the language is most commonly used in constructing wireless sensor network applications and is emerging as the development standard in this domain. There are hundreds (or perhaps thousands) of documented projects, both academic and commercial, involving the use of this specialized language.

nesC applications are structured as collections of modules that provide functionality through well-defined interfaces. Each module encapsulates state and behavior in a manner analogous to an object-oriented class. There are, however, important differences between modules and classes. Consider two of the most important: First, modules are constructed statically; they behave as singleton instances. Second, modules are completely decoupled; they never reference one another directly. Instead, each module specifies the set of functions required to realize its behavior through the declaration of its required interfaces. Hence, nesC modules define the interfaces they provide, as well as the interfaces they use.

In addition to declaring commands, functions invoked by using modules, an interface may declare events. An event declaration imposes an implementation responsibility on modules that use the declaring interface. Specifically, modules using the interface are required to implement an event handler with the specified signature. This handler is invoked in response to events signaled by the interface provider. Hence, as in C# [12], nesC interfaces are bidirectional.

Another distinct feature of nesC is its support for tasks, a lightweight alternative to threads. A task is a module-private function posted for later execution by the TinyOS task scheduler. Tasks are typically used for long-running operations or operations that must be executed outside interrupt context. A common nesC idiom involves the execution of a long-running unit of work (e.g., data transmission, sensor sampling) initiated through a command. The command starts the work in hardware and immediately returns control to the caller. When the relevant hardware component signals completion, the corresponding interrupt handler posts a task, which, when executed, signals completion of the work on the initiating module (outside interrupt context). Another common idiom involves a collection of tasks that break a large processor-bound operation into smaller units of work.

In addition to modules, nesC applications are composed of special components referred to as configurations. Each configuration maps the interfaces used by a specified set of modules to modules that provide those interfaces. Syntactically,
this map takes the form of a list of participating modules and a series of wiring statements that bind their interfaces. It is worth noting that multiple providing modules may be bound to the same used interface. Invocations placed through the interface are then dispatched to all of the bound providers.

This section provides only a short overview of nesC and TinyOS; a full description of the platform is beyond the scope of this article. It is, however, intended to show that systems developed using this platform are fundamentally different from systems developed using traditional imperative languages. The following points summarize the key differences: Unlike programs written in standard imperative languages, these systems consist of short-lived, reactive operations executed in response to environmental stimuli (e.g., sound, magnetic field changes). The systems provide no blocking function calls; long-running operations are deferred to tasks. Implementations are fully decoupled; modules use and provide interfaces, and the system is wired together independent of their implementations. Multiple modules can be wired to a single interface, resulting in the fanning of function calls and returns. All of these features introduce unique challenges for an analysis and instrumentation toolkit, and ours is the first and only system capable of providing these services for this unique development platform.

3 Toolkit Design and Implementation

The nesC Analysis and Instrumentation Toolkit (nAIT) provides a foundation for extending the use of automated software engineering methods to the domain of wireless sensor networks. The following subsections describe our design goals and the implementation of the toolkit.

3.1 Design Goals

The toolkit design is guided by two main goals. First, the toolkit must enable users to construct new language processing tools that operate on programs written for any hardware or simulator platform capable of running nesC applications. 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 representations of programs, (2) modify existing programs programmatically, and (3) generate new programs and program segments programmatically.

The nAIT design achieves each of these design goals. First, the toolkit is hardware platform independent. One approach to performing instrumentation is
to modify binary executables. This approach, however, is specific to the underlying hardware architecture. Hence, to support a wide range of targets, the toolkit is source-based. The source-based approach enables users to apply the toolkit directly to nesC source files before any platform-specific transformations are performed by the compiler chain.

Second, the toolkit simplifies the development of new software engineering tools by providing an API that defines methods for easily traversing, modifying, and generating in-memory representations of programs, and a graphical tool for visualizing these representations. In addition to supporting traditional visitor-based traversals [14], the API provides methods that enable developers to quickly locate elements within a representation by their respective types. Developers can, for instance, use the API to find all functions defined within a module. The API also provides methods that enable developers to perform common modifications to an in-memory program representation. For example, the API contains a method that enables developers to add a wiring statement to a nesC configuration. Finally, the API provides methods that enable developers to create complex program segments without knowledge of the requisite representation structure. Developers can, for example, use the API to create an in-memory representation of a function call using standard calling syntax, rather than building the corresponding object trees manually. The graphical tool helps developers understand the composition of the in-memory program representation and serves as a guide for developers as they create their own tools. Users can, for instance, use the tool to visualize the structure of existing applications. By inspecting these visualizations, users are better able to understand, for example, how the in-memory representation of a function call is organized. This understanding in turn helps the users to identify function calls programmatically.

Using the toolkit, developers are able to quickly and easily develop language processing tools for nesC. Two examples of such tools are found in Section 4. These new software engineering tools will improve the development and maintenance of systems created using this unique language and platform.

3.2 Toolkit Implementation

We now turn our attention to the implementation of the toolkit. The steps involved in applying the toolkit are summarized in Figure 1. In the figure, solid arrow heads represent data flow through the system, while open arrow heads represent client interactions with the system. The first step in applying the toolkit is to scan and parse the source base of the target application. The

\footnote{We use the term client to refer to software components that use the API to implement their functionality.}
Fig. 1. Analysis & Instrumentation Process Overview

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 the structure of the ASTs, or the ASTs can be exposed to external software engineering tools via an API interface. These tools use the API to perform the specific analysis and instrumentation tasks desired for that tool. If a 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.

Implementation Technologies. Third party tools were used to construct the scanner and parser. These tools include JFlex [26], a Java-based scanner generator, and CUP [21], 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 [4], 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 regeneration. Specifically, the grammar was modified to support the #include preprocessor directive. Preprocessor directives are traditionally eliminated prior to parse time; however, preprocessing nesC source files results in programs that contain duplicated symbols at compile time. To address this problem, we extract and apply all #define macros, integrate the symbols defined in the included files into the symbol table, and retain the #include statements in the grammar.

To simplify code traversal and modification, the parser performs several AST-level transformations while processing each source file. The goal of these transformations is to normalize the input program. Where possible, syntactic variation (from an idealized implementation) is eliminated to provide tool developers with a uniform foundation for implementing analysis and instrumentation
functions. The following paragraphs describe each of the transformations applied, as well as the benefits that they provide.

**Compound Statement Normalization.** The toolkit adds explicit compound statements for all control structures, simplifying code insertion tasks. Without the transformation, inserting a line of code within a control structure requires that the body of the structure be a compound statement; otherwise, a new compound statement must be created.

**Return Statement Normalization.** The toolkit inserts an explicit `return` statement at the termination point(s) of all `void`-returning functions, simplifying the task of identifying and navigating to function exit points. In particular, the transformation eliminates the need for control-flow analysis; a simple search for `return` statements suffices.

**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, simplifying the task of distinguishing interface users from interface providers. Without the transformation, identifying the set of components that use or provide a given command requires consideration of the binding direction of each wiring statement. (Recall that the wiring operator always “points” from the user of an interface to the provider.)

**Interface Name Normalization.** The toolkit substitutes fully-qualified interface names within all wiring statements. For example, a wiring statement of the form `A → B` would be transformed to `A.I → B.I`. The transformation simplifies the identification of bound interfaces. Without it, identifying the interface bound by a given wiring statement requires searching the set of available interfaces and identifying a valid match (i.e., a basic type inference routine).

**Component/Interface Reference Normalization.** The toolkit explicitly renames all component references within configurations, as well as all interface references within module implementations. This simplifies component and interface search and matching routines. Without the transformation, special checks are necessary to determine whether an interface or component is renamed when searching for instances of the element.

**List Normalization.** Finally, the toolkit inserts a *dummy* node in each empty list within an AST. For example, the AST corresponding to a top-level configuration (i.e., a configuration that neither uses nor provides interfaces), will include an empty list node rather than a null reference. Introducing these nodes not only simplifies list traversal by eliminating special-case logic, but provides a target object for AST navigation. (Recall that `getNodesOfType()` recursively searches for objects of a specified class type.) Without the transformation, navigating to a particular set of lists requires a more manual (and error-prone) search process.
These transformations support our goal of enabling developers to easily create new software engineering tools. Without normalization, analysis and instrumentation tasks are still possible, but users would be required to contend with more syntactically diverse ASTs, requiring more special-case checks throughout their code.

**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\(^2\) 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. Tables 1, 2, and 3 include representative lists of these methods\(^3\), each of which addresses the underlying design goals.

First, Table 1 lists four representative traversal methods. `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 uses the specified component name (i.e., the first ConfigComp found in the mapping that corresponds to a configuration component that contains a reference to the component with the specified name). This method is useful, for example, when a new 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 use 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 are achieved by passing 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

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Method Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConfigComp</td>
<td><code>findConfigUsing(String compName, Map sourceFiles)</code></td>
</tr>
<tr>
<td>List</td>
<td><code>findAllConfigsUsing(String compName, Map sourceFiles)</code></td>
</tr>
<tr>
<td>AstNode</td>
<td><code>getParent()</code></td>
</tr>
<tr>
<td>List</td>
<td><code>getNodesOfType(Class type, boolean recursive)</code></td>
</tr>
</tbody>
</table>

Table 1
Traversal Methods (partial)

---

\(^2\) Conceptually, the result is a set; however, to simplify search tasks, the method returns a mapping from file name (component/interface name) to AST.

\(^3\) Some of the type and function names have been shortened for presentation.
public abstract class AbstractAstNode implements AstNode {

    public <Type extends AstNode> List<Type> getNodesOfType(Class<Type> type, boolean recursive) {
        List<Type> nodeList = ...;
        for (all non-static fields of this class) {
            Let “fieldObject” be the object for the current field
            if (fieldObject != null) {
                if (type.isAssignableFrom(fieldObject.getClass())) {
                    nodeList.add((Type) fieldObject);
                }
                if (recursive && fieldObject instanceof AstNode) {
                    nodeList.addAll((AstNode) fieldObject).getNodesOfType(type, recursive));
                }
                if (fieldObject is an array or a Collection) {
                    for (Object o : fieldObject) {
                        if (type.isAssignableFrom(o)) {
                            nodeList.add((Type) o);
                        }
                        if (recursive && o instanceof AstNode) {
                            ((AstNode) o).getNodesOfType(type, recursive));
                        }
                    }
                    return nodeList;
                }
            }
        }
        return nodeList;
    }

Listing 1. Pseudocode for getNodesOfType()

containing context associated with any AST node.

The final traversal method in the table, getNodesOfType(), is especially interesting. The method recursively searches for AST nodes of a specified 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 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). Each of these methods enable users to easily traverse the in-memory representations of nesC programs.

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 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

---

4 We emphasize that the listing presents a simplified version of the method implementation; many details have been omitted.
<table>
<thead>
<tr>
<th>Return Type</th>
<th>Method Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>addCompToConf(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>

Table 2
Modification Methods (partial)

types that extend AstNode (Line 2). Next, the method iterates over all of the non-static fields defined in the activated object’s class type (Lines 7 – 27). For each non-null field, the method tests to determine if the field is of the desired type (Lines 11 – 13). If the object is of the desired type, the method adds the object to the result list. If the recursive flag is set, the method is called recursively on the associated field (Lines 14 – 17). If the field is not of the desired type, a check is made to determine whether the object is an array or a Collection (Line 18). If so, the method iterates over each element (Lines 19 – 26). Again, if the type of the element matches the desired type, the method adds the sub-element to the result list (Line 21), and if the recursive flag is set, the method is called recursively on the object (Lines 23 – 26).

Second, Table 2 lists five representative modification methods. addComponentToConf() (abbreviated in the table) is used to introduce a new component reference in an existing configuration. The method takes the component to which to add the new reference, the (actual) name of the referenced component, and a local name for the component. The method is useful, for example, when injecting new services into existing applications (e.g., a time synchronization component). The addUses() method 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. The method is useful, for example, when instrumenting a component and the newly injected code requires additional services (provided by unreferenced components). addProvides() is used to introduce a new provided interface within a component, and is similarly useful when instrumenting a component to provide additional services (which must be accessed from other components). 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. After a component has been added to a configuration using addComponentToConf(), addWiring() is useful for adding wirings to the newly introduced component.

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 components and generic configurations. The approach is to
duplicate generic types 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 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. The method is useful because it provides tool developers an analysis and instrumentation model that mirrors the semantics of the underlying programming model. In short, it enables developers to analyze and modify the products of the instantiation process. Each of the modification methods directly support our goal of simplifying the programmatic modification of nesC programs.

Finally, Table 3 lists eight representative generation methods. These methods are similar to the the methods provided by the CodeDOM [35] API included with C#. `generateExprStmt()` accepts a string containing a nesC expression statement and returns an AST corresponding to that statement. The method is useful, for example, when 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. The `generateIfStmt()` method accepts a string containing an integer 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 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 reference 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, `generateModule()` accepts a module name and returns an AST corresponding to an empty module with that name. Rather than simply mod-

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Method Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExprStmt</td>
<td><code>generateExprStmt(String expressionString)</code></td>
</tr>
<tr>
<td>JustDatadef</td>
<td><code>generateEnum(String enumerationString)</code></td>
</tr>
<tr>
<td>Simplef</td>
<td><code>generateIfStmt(String conditionString)</code></td>
</tr>
<tr>
<td>JustDatadef</td>
<td><code>generateVarDec(String varDecString)</code></td>
</tr>
<tr>
<td>Fndef</td>
<td><code>generateFunctionDef(String signature)</code></td>
</tr>
<tr>
<td>ComponentRef</td>
<td><code>generateComponentRef(String componentRefString)</code></td>
</tr>
<tr>
<td>Connection</td>
<td><code>generateWiring(String wiringString)</code></td>
</tr>
<tr>
<td>ModuleComponent</td>
<td><code>generateModule(String moduleName)</code></td>
</tr>
</tbody>
</table>

Table 3
Generation Methods (partial)
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 2. Sample Generation Method

ifying an existing AST, this method can be used to programmatically create new modules. All of the generation methods make it easier for users to programmatically generate ASTs that can be inserted into an existing source base, simplifying instrumentation tasks.

As noted previously, the toolkit provides methods that simplify the most common analysis and instrumentation tasks. Users may need to develop custom generation methods to meet their needs. Listing 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 within 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. 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 (Lines 8 – 10). The method then uses getNodesOfType() to access a list of objects corresponding to the code segment(s) of interest (Lines 12 – 13). Finally, the method returns the AST corresponding to the desired code segment (Line 15). The caller of this method might 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 nesC scanner and parser. Additionally, getNodesOfType() provides an easy way to navigate to the appropriate sub-tree within the (larger) AST produced by the parser. Whether using existing methods or developing custom methods, the nAIT design makes it easier for users to traverse, modify, and generate in-memory representations of nesC programs.

nesC interfaces, modules, and configurations are also valid parse targets.

In this case, a list containing only one element.
The final component of the toolkit is a visualization tool for exploring ASTs. The AST Explorer is built on JUNG [38], a Java-based graph rendering library. The tool takes as input an AST node and displays a graphical representation of the AST rooted at that node. The tool uses the traversal methods provided by the nAIT to walk the AST and generate the visualization.

Figure 2 includes a sample screen capture of the AST Explorer. The figure shows a portion of an AST corresponding to a module-level 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. Labels associated with generic AST nodes, such as Pair, include the names of the actual type arguments or null, as appropriate. Leaf nodes are labeled using the text that appears in the target source file. In the figure, uint8_t and data are leaf nodes corresponding to the type and name of the variable in the source, respectively. Edges in the tree are labeled using the corresponding variable names within the containing AST nodes. For example, the JustDatadef class contains two fields, typeElementList and declaration.

The buttons at the bottom of the figure allow users to control the behavior of the visualization. The “collapse” button is used to filter out unwanted information by collapsing a subtree rooted at a selected node. The “expand” button supports the opposite behavior. The “toggle mode” button is used to switch between transform and selection modes. Transform mode allows users to move the graph (as a whole) within the window. Selection mode allows
users to select and move individual nodes within the graph.

The AST Explorer supports comprehension of AST structures across the development process. For example, when a user is interested in traversing an AST to identify 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 tool helps the user to understand the node containment hierarchy. This information is useful in determining which AST traversal technique is most appropriate. When modifying an AST, the user again relies on the AST Explorer to visualize a similar code segment. Visualizing the AST structure helps in identifying the set of classes involved in the modification task. Finally, when developing a new generation method, the user relies on the AST Explorer to determine where in the parse target the subtree of interest is contained.

4 Use-Case Scenarios

We now consider two use-case scenarios that illustrate the features and benefits of the nAIt design and outline how the toolkit’s traversal, modification, and generation methods are used to develop new source-based software engineering tools for nesC. The first use-case describes the development of a source-based state predicate evaluation tool. The second describes the development of a runtime trace recording and visualization tool.

4.1 Predicate Evaluator

The *Predicate Evaluator* allows developers to monitor up to 3 state predicates within a nesC program to detect invariant violations. Each predicate is bound to an LED on the hosting device. *Off* indicates that the predicate has *always* evaluated to true, while *on* indicates that the predicate has, *at least once*, evaluated to false. The tool presents users with an interface for selecting the top-level configuration of the target application, selecting the variables associated with each predicate, and defining the associated predicates in terms of the selected 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\(^7\) 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

\(^7\) Throughout this section, when it is unnecessary to distinguish between nesC commands, events, tasks, and local functions, we simply use the term *function*. 
Fig. 3. Predicate Evaluator

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 the nAIT’s traversal, modification, and generation methods were used to simplify the tool’s implementation.

**Tool Overview.** The process begins with the application window shown in Figure 3. 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 has been selected, the tool uses the nAIT to parse the application source materials. Next, all of the resulting ASTs are searched to identify the constituent modules, program variables, and associated type data. Once this information has been collected, the “program symbols” list on the left side of the window displays 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. 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 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 an LED has been selected, the user selects one or more variables from the “program symbols” list. A pointer to each selected variable is then available when defining the associated predicate. In the figure, HeapP.available, HeapP.capacity, and TestP.used have been selected. Next, the user defines the predicate as a boolean expression defined in terms of the selected vari-

---

8 The “program symbols” list excludes volatile variables.
ables. 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 exist: 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.

When all of the required predicates have been defined, the “save” button is used to initiate the instrumentation process. The tool then uses the 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.

**Tool Implementation.** The Predicate Evaluator was built on the nAIT and its implementation was guided by the AST Explorer. Here we outline the implementation and discuss portions of the development process.

When a user selects a top-level configuration, the Predicate Evaluator uses the nAIT to perform a configuration-based parse of the nesC source materials rooted at the selected file. This process returns a list of ASTs, each corresponding to a parsed file. Next, the Predicate Evaluator searches the list for ASTs that correspond to nesC modules using `getNodesOfType()`. For each match, `getNodesOfType()` is again used to identify the variable declarations within the module. Traditional accessor methods are then used to collect type information. (The aggregated data is later used to populate the window’s selection list.) The method that realizes this behavior in the Predicate Evaluator consists of only 49 lines of code.

During the development process, we used the *AST Explorer* to understand how type information is represented within ASTs. Figure 4 shows a screen capture of an AST Explorer window that includes the subtree corresponding to the declaration of a module-level 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 list in the figure contains only one entry, a `typenameTypeElement`. This class represents 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 `InitDecs`, a declaration that may include an initializer. The declaration consists of a generic `Pair`, associating a list of attributes with each declaration. In the figure, the declared variable contains no attributes, so the attribute list is empty. The second part of the `Pair`, `Initdcl`, corresponds to a single declaration. 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 exist in the associated AST to represent that information. In the case of this simple variable declaration, the AST consists of 9 nodes; more complex expressions consist of dozens. The AST Explorer supports AST comprehension, providing structural insights that would otherwise be difficult to glean.

After the user clicks the “save” button, the nAIT’s generation methods are used to create three new nesC source files that encapsulate the predicate evaluation logic. The first, `PredicateEvaluator.nc`, defines the nesC interface provided by the main predicate evaluation component. The process of adding each constituent command to this interface involves only three lines of code — a call to a generation method, plus local AST calls to integrate the generated code within the interface. The generated interface provides commands for registering (the address of) each user-selected state variable, as well as commands for reevaluating the generated predicates. The second generated source file, `PredicateEvaluatorM.nc`, implements this interface; it is generated using `generateModule()`. The implementation consists of state variables used
to store pointers to the selected variables, and implementations of the commands defined by `PredicateEvaluator`. The `generateVarDec()` and `generateFunctionDef()` methods are used to generate the variables and commands, respectively. The user-defined predicates are used to implement the bodies of the corresponding commands. If a predicate evaluates to false, the implementation activates the corresponding LED. The final generated source file is `PredicateEvaluatorC.nc`. The file defines a nesC configuration that re-exposes the `PredicateEvaluator` interface provided by `PredicateEvaluatorM` and wires-in the standard implementation of the `Leds` interface. The implementation of the method to generate this file consists of only 12 lines of code. Finally, the (disk) files are generated using the nesC source regeneration visitor included with the nAIT.

Next, the application source base must be instrumented to register the user-selected state variables with `PredicateEvaluatorM`: First, each module containing a selected variable is modified to use the generated `PredicateEvaluator` interface using an appropriate generation method and a call to `addUses()`. Each module is similarly modified to use the `Boot` interface, appropriately renamed to avoid conflicts. The `Boot.booted()` event, generated using `generateFunctionDef()`, is populated with calls to register the selected variables using `generateExprStmt()`. The method responsible for this behavior consists of only 18 lines of code.

Next, each function that modifies a selected variable must be instrumented to reevaluate the affected predicate(s): First, `getNodesOfType()` is used to identify the functions within each module. The method is then used again to search for affecting assignment, increment, and decrement statements/expressions\(^{11}\). For each match, `getNodesOfType()` is used for a third time to identify all return statements within the function. (Recall that the nAIT automatically includes explicit returns at parse time.) `generateExprStmt()` is then used to generate calls to the appropriate reevaluation commands exposed by `PredicateEvaluator`, and local AST methods are used to insert the generated calls immediately prior to each `return` statement.

Finally, the application’s configurations must be updated to wire the `PredicateEvaluator` interface to the `PredicateEvaluatorC` component for each affected module. This final step is analogous to the preceding steps; we omit the details.

We emphasize that the nAIT significantly simplified the implementation of the Predicate Evaluator — a reasonably complex analysis and instrumentation tool. Indeed, the complete implementation consists of approximately 1500 lines, including comments and whitespace. The majority of this code is associated with the creation and management of the graphical user interface. The

\(^{11}\) This approach only supports direct modifications; modifications via indirection are not considered.
complexity associated with the nesC language is hidden by the nAIT, and implementation of the Predicate Evaluator focuses on the software engineering task at hand.

4.2 Control Flow Visualizer

The Control Flow Visualizer supports the visualization of static and dynamic control flow paths within a nesC application. Static (i.e., potential) paths are represented using simple directed graphs. Dynamic (i.e., actual) paths are represented using UML sequence diagrams. The constituent components of the implementation include an interface for injecting logging probes, a lightweight logging service, tools for extracting logged trace data, and the aforementioned visualization interfaces. The remainder of the subsection outlines the use of the tool and describes how the nAIT was used to simplify portions of its implementation.

**Tool Overview.** The steps involved in applying the Control Flow Visualizer are summarized in Figure 5. The first step uses the nAIT to analyze and transform the input system. The result is an instantiated source base and metadata detailing program symbols and function calling relationships. The metadata may be used to generate an annotated call graph for the system, or serve as input to a second step, in which a developer selects a set of functions to be traced (or “probed”). The result of this step is an instrumented source base that can be compiled and executed. Runtime execution data is then extracted and used to generate a sequence diagram that describes the program run.

**Tool Implementation.** The visualization process begins with a configuration-based parse originating from the configuration path passed as argument by the user. As before, this results in a list of ASTs corresponding to the target sys-

---

12 A full description of the Control Flow Visualizer is beyond the scope of this article; the interested reader is directed to [11] for a complete description of the system.

13 The figure is a simplified version of a more complete representation presented in [11].
tem’s source files. The tool then iterates over the ASTs to identify program modules and their constituent functions using getNodesOfType(). A mapping between modules and functions is stored for later use in populating the selection interface used to inject logging probes.

Next, instantiateGenerics() is used to instantiate the source base. The resulting AST set is then analyzed to gather the information necessary to construct the system’s (static) call graph. After identifying all module instances, getNodesOfType() is used to identify the call sites within the constituent functions. Each call site is then examined to identify the called interface. Next, the tool searches the AST set for configurations, using getNodesOfType() to identify the constituent components and wiring statements. These statements are inspected to identify the target site bound to each call site, completing the metadata required for call graph construction.

The more interesting case involves sequence diagram construction: The application provides a simple interface for selecting probe injection points, populated using the module-to-function mapping created at startup. After the injection points have been selected, the tool iterates over the ASTs, identifying trees corresponding to nesC modules. If an AST corresponds to a module that contains a selected function, addUses() is used to add the TraceRecorder interface to the module’s uses list. This interface exposes the commands used to record trace events. getNodesOfType() is then used to identify the module’s functions; selected functions are modified to include entry and exit probes.

The basic instrumentation procedure is illustrated in Listings 3 and 4\textsuperscript{14}. The

\textsuperscript{14}These listings are adapted from [11].
first shows a simple nesC command prior to instrumentation; the second shows equivalent instrumented code. First, the command is modified to include a call to the `enter()` command provided by the `TraceRecorder` interface. The tool uses `generateExprStmt()` to generate an appropriate AST corresponding to the call. Next, local AST methods are used to wrap the body of the function within an anonymous block and to place the call to `enter()` before that block. Next, each exit point is modified to include a call to `exit()` prior to return; `getNodesOfType()` is used to identify all AST nodes corresponding to `return` statements. If a `return` statement has no associated expression (i.e., if the function returns no value), `getParent()` is used to access the containing context of the `return`, and `generateExprStmt()` is again used to generate a call to `TraceRecorder.exit()`. Local AST methods are then used to insert the newly created AST node prior to the `return` statement. If a `return` statement has an associated expression, the expression is wrapped within a gcc statement expression\(^{15}\), allowing the exit event to be captured after any function calls that might appear in the `return` expression. A combination of `generateExprStmt()` and local AST methods are used to generate the statement expression. Finally, the return statement’s expression is updated to reflect the newly created expression-statement.

Note that each probe records the `instanceId` used to identify the containing component. This identifier is introduced as a module-level enumeration constant after all selected functions have been instrumented: `getNodesOfType()` is used to identify the list of AST nodes corresponding to module-level variable, type, and function prototype declarations. `generateEnum()` is used to generate the enumeration, assigning `instanceId` a unique integer value. Finally, local AST-level methods are used to insert the enumeration into the AST list.

Next, each instrumented module must be wired to a realization of the `TraceRecorder` interface. For each such module, `findConfigUsing()` is used to identify the configurations that use the module. On each match, `addComponentToConf()` is used to add `TraceRecorderC`, a component that realizes the `TraceRecorder` interface, to the configuration’s list of components. The new component is then wired to the `TraceRecorder` interface using `generateWiring()` and `addWiring()`. Finally, the source regeneration visitor is used to generate the instrumented source materials.

Again, we emphasize that the nAIT design significantly simplified the development of the Control Flow Visualizer — a complex program comprehension tool. The key observation is that the toolkit hides the complexity of the nesC language and provides high-level methods that handle the most common analysis and instrumentation tasks.

\(^{15}\)A statement expression is a compound statement enclosed in parenthesis whose value is the value of the last expression in the block.
5 Quantitative Analysis

We now examine the nAIT’s resource requirements. We focus on the time required to load, traverse, and instantiate (in the case of generics) ASTs. We also study the memory required to represent ASTs. Applications included as part of the standard TinyOS distribution are used as the test suite.

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 the Sun 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.

5.1 Time Requirements

We consider the time required to (i) parse programs, (ii) use traversal methods to gather static information about programs, and (iii) instantiate generic components.

Parse Time. The parse time of the nAIT plays an important role in assessing its utility. If the time required to parse a compilation unit is too large, the toolkit may not be suitable for many applications. To evaluate the speed of the parsing system, we developed a test application that measures the time required to parse the source files within an input source base. The measurement approach accounts for file dependencies: The parse time associated with a file \( f \) is calculated by subtracting the time required to parse the dependencies of \( f \) from the total time required to parse \( f \) (and its dependencies). Note that these times include the time required to scan the filesystem, run the C preprocessor, and perform the normalization procedures described in Section 3.2.

A summary of the experimental results is shown in Table 4. Each row corresponds to an input program. The Files column represents the number of files scanned and parsed, including files that consist of only preprocessor macros, which are not represented in ASTs. The remaining columns have the obvious meanings. Despite the large variation in parsing time observed during each experiment, the results are favorable. In the worst case, the toolkit requires approximately 21 seconds to scan and parse the largest project, MultihopOscilloscopeLqi. This makes the toolkit well-suited for analysis and instrumentation activities that do not require the faster modify-and-execute cycles expected from compilers.
Table 4
Source File Scanning/Parsing Times

<table>
<thead>
<tr>
<th>Application</th>
<th>Files</th>
<th>Time (seconds)</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Min</td>
<td>Max</td>
<td>Median</td>
<td>Mean</td>
<td>Stddev</td>
</tr>
<tr>
<td>BaseStation</td>
<td>197</td>
<td>0.000316</td>
<td>1.777482</td>
<td>0.009829</td>
<td>0.073345</td>
<td>0.205816</td>
</tr>
<tr>
<td>Blink</td>
<td>71</td>
<td>0.000387</td>
<td>0.527205</td>
<td>0.013929</td>
<td>0.059630</td>
<td>0.106070</td>
</tr>
<tr>
<td>MViz</td>
<td>252</td>
<td>0.000335</td>
<td>2.174376</td>
<td>0.010937</td>
<td>0.074172</td>
<td>0.239981</td>
</tr>
<tr>
<td>MultihopOscilloscope</td>
<td>278</td>
<td>0.000311</td>
<td>2.374736</td>
<td>0.011520</td>
<td>0.073912</td>
<td>0.228117</td>
</tr>
<tr>
<td>MultihopOscilloscopeLqi</td>
<td>267</td>
<td>0.000289</td>
<td>2.259113</td>
<td>0.010000</td>
<td>0.081133</td>
<td>0.235193</td>
</tr>
<tr>
<td>Null</td>
<td>41</td>
<td>0.000387</td>
<td>0.494584</td>
<td>0.005332</td>
<td>0.067967</td>
<td>0.113593</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>211</td>
<td>0.000299</td>
<td>1.895630</td>
<td>0.012670</td>
<td>0.074279</td>
<td>0.216454</td>
</tr>
<tr>
<td>Powerup</td>
<td>51</td>
<td>0.000388</td>
<td>0.502854</td>
<td>0.014673</td>
<td>0.063928</td>
<td>0.109360</td>
</tr>
<tr>
<td>RadioCountToLeds</td>
<td>183</td>
<td>0.000288</td>
<td>1.603554</td>
<td>0.011901</td>
<td>0.085674</td>
<td>0.239886</td>
</tr>
<tr>
<td>RadioSenseToLeds</td>
<td>211</td>
<td>0.000284</td>
<td>1.882680</td>
<td>0.011658</td>
<td>0.078553</td>
<td>0.206830</td>
</tr>
<tr>
<td>Sense</td>
<td>107</td>
<td>0.000388</td>
<td>0.843370</td>
<td>0.012847</td>
<td>0.067059</td>
<td>0.133331</td>
</tr>
</tbody>
</table>

Traversal Time. The traversal time of the nAIT represents the time required to perform common traversal operations on ASTs and is again important in assessing the toolkit’s utility. To evaluate traversal time, we developed a test application to measure the time required to traverse NesC programs 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 5 and Figure 6. In the table, the Modules column represents the time required to identify the names of the modules within the input system. The Functions column represents the time required to identify the names of the modules within the system, as well as the names of all of the functions defined within those modules. Finally, the Calls column represents the time required to identify the names of the modules, the names of the functions within the modules, and the names of the functions called by each of the function bodies. As highlighted in the figure, the time required to identify module names is negligible; the corresponding bars are barely visible. This is due to the fact that the information required to determine whether an AST represents a module is stored at the root of the AST – a deep traversal is unnecessary. Identifying the functions within the modules and the calls within each function requires more time. We note that MultihopOscilloscope, at approximately 3 seconds, required the most time to process\(^\text{16}\). This result indicates that the toolkit is well-suited for analysis and instrumentation activities that involve the traversal of module ASTs.

Instantiation Time. The instantiation time of the nAIT represents the time

\(^{16}\)Recall from Table 4 that MultihopOscilloscope is the program with the largest number of files.
Table 5
AST Traversal Times

<table>
<thead>
<tr>
<th>Application</th>
<th>Time (seconds)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Modules</td>
<td>Functions</td>
<td>Calls</td>
</tr>
<tr>
<td>BaseStation</td>
<td>0.003013</td>
<td>1.019369</td>
<td>2.135666</td>
</tr>
<tr>
<td>Blink</td>
<td>0.001451</td>
<td>0.235980</td>
<td>0.418365</td>
</tr>
<tr>
<td>MViz</td>
<td>0.003639</td>
<td>1.398710</td>
<td>2.588664</td>
</tr>
<tr>
<td>MultihopOscilloscope</td>
<td>0.004131</td>
<td>1.633255</td>
<td>3.052278</td>
</tr>
<tr>
<td>MultihopOscilloscopeLqi</td>
<td>0.004104</td>
<td>1.461011</td>
<td>2.762541</td>
</tr>
<tr>
<td>Null</td>
<td>0.000980</td>
<td>0.127172</td>
<td>0.224839</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>0.003751</td>
<td>1.064990</td>
<td>2.034187</td>
</tr>
<tr>
<td>Powerup</td>
<td>0.001146</td>
<td>0.160780</td>
<td>0.282425</td>
</tr>
<tr>
<td>RadioCountToLeds</td>
<td>0.002782</td>
<td>0.816394</td>
<td>1.582806</td>
</tr>
<tr>
<td>RadioSenseToLeds</td>
<td>0.003549</td>
<td>1.052735</td>
<td>1.988128</td>
</tr>
<tr>
<td>Sense</td>
<td>0.001938</td>
<td>0.465612</td>
<td>0.979849</td>
</tr>
</tbody>
</table>

A summary of the experimental results is shown in Table 6. The File Count column represents the number of files scanned/parsed. The Inst. Count column represents the number of ASTs resulting from the call to instantiateGenerics(). The New Files column captures the difference between the first two columns,
Table 6
Generic Instantiation Times

<table>
<thead>
<tr>
<th>Application</th>
<th>File Count</th>
<th>Inst. Count</th>
<th>New Count</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>

i.e., the number of new ASTs introduced by the instantiation process. Finally, the Time column represents 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. By contrast, Null made use of no generic components and took less than 2 seconds to process. We believe the time required to instantiate generic components is acceptable for most applications.

5.2 Space Requirements

We now consider the memory requirements associated with the nAIT. Specifically, we consider the representation sizes of (i) interfaces, (ii) components, (iii) modules, and (iv) header files. To interpret the results, it is useful to note that there are 159 classes used to represent AST nodes. Considering these classes, (i) the minimum object size is 24 bytes; (ii) the maximum object size is 48 bytes; (iii) the average object size is 25.66 bytes; and (iv) the standard deviation among the object sizes is 4.12 bytes. We note that the profiling tool used to gather the results limits size accuracy to ± (±/−) 500 bytes for objects larger than 9000 bytes.

Interface Space Requirements. Table 7 summarizes the space requirements associated with ASTs corresponding to nesC interfaces. The Files column represents the number of files that correspond to interfaces in the respective applications. The other columns have the obvious meanings.

The minimum AST size for an interface ranges between 1064 bytes and 1104 bytes. The maximum AST size ranges between 10500 and 54500 bytes. The
<table>
<thead>
<tr>
<th>Application</th>
<th>Files</th>
<th>Min</th>
<th>Max</th>
<th>Median</th>
<th>Mean</th>
<th>Stddev</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseStation</td>
<td>73</td>
<td>1080</td>
<td>54500</td>
<td>4440.00</td>
<td>5908.33</td>
<td>7531.61</td>
</tr>
<tr>
<td>Blink</td>
<td>20</td>
<td>1064</td>
<td>11500</td>
<td>5212.00</td>
<td>5239.20</td>
<td>3738.39</td>
</tr>
<tr>
<td>MViz</td>
<td>89</td>
<td>1064</td>
<td>11500</td>
<td>6065.98</td>
<td>7166.74</td>
<td>7531.61</td>
</tr>
<tr>
<td>MultihopOscilloscope</td>
<td>96</td>
<td>1096</td>
<td>54500</td>
<td>4508.00</td>
<td>4520.00</td>
<td>6065.98</td>
</tr>
<tr>
<td>MultihopOscilloscopeLqi</td>
<td>91</td>
<td>1104</td>
<td>54500</td>
<td>4520.00</td>
<td>5925.63</td>
<td>6981.77</td>
</tr>
<tr>
<td>Null</td>
<td>13</td>
<td>1064</td>
<td>10500</td>
<td>2136.00</td>
<td>3922.15</td>
<td>3252.23</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>73</td>
<td>1080</td>
<td>54500</td>
<td>4496.00</td>
<td>6212.00</td>
<td>7651.62</td>
</tr>
<tr>
<td>Powerup</td>
<td>16</td>
<td>1072</td>
<td>11500</td>
<td>4264.00</td>
<td>5018.25</td>
<td>3788.38</td>
</tr>
<tr>
<td>RadioCountToLeds</td>
<td>65</td>
<td>1088</td>
<td>54500</td>
<td>4504.00</td>
<td>6202.15</td>
<td>7906.09</td>
</tr>
<tr>
<td>RadioSenseToLeds</td>
<td>73</td>
<td>1088</td>
<td>54500</td>
<td>4504.00</td>
<td>6221.42</td>
<td>7649.91</td>
</tr>
<tr>
<td>Sense</td>
<td>35</td>
<td>1064</td>
<td>16500</td>
<td>3944.00</td>
<td>5053.49</td>
<td>3904.77</td>
</tr>
</tbody>
</table>

Table 7
Interface Source File AST Sizes

minimum and maximum values are shared among several of the applications; the corresponding interfaces are core to TinyOS and used by many of the input applications. The minimum size of 1064 bytes, shared by 4 of the applications, corresponds to the file $TOSDIR/interfaces/Boot.nc, an interface that defines only a single operation, namely event void booted(). Similarly, the maximum size of 54500 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 a high degree of variability in AST size for each application, as indicated by the large standard deviations. We therefore consider not only the mean, but also the median sizes of the ASTs. In MultihopOscilloscope, the application with the largest number of interfaces, the mean-based estimated memory consumed is 591302.4 bytes (approximately 577kB). The median-based estimated memory consumed is 432768.0 (approximately 423kB). We believe that this is an acceptable figure for most applications of the nAIT.

Configuration Space Requirements. Table 8 summarizes the space requirements associated with ASTs corresponding to nesC configurations. The minimum size of a configuration AST ranges from 2008 bytes to 2376 bytes. The maximum size ranges from 68500 bytes to 132500 bytes. As with interfaces, there are a number of applications that share the same minimum and maximum sizes; again, 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 133500 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 state-
<table>
<thead>
<tr>
<th>Application</th>
<th>Files</th>
<th>Min</th>
<th>Max</th>
<th>Median</th>
<th>Mean</th>
<th>Stddev</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseStation</td>
<td>56</td>
<td>2352</td>
<td>133500</td>
<td>13024.14</td>
<td>8500.00</td>
<td>13024.14</td>
</tr>
<tr>
<td>Blink</td>
<td>19</td>
<td>2344</td>
<td>132500</td>
<td>16094.74</td>
<td>30927.04</td>
<td>17732.80</td>
</tr>
<tr>
<td>MViz</td>
<td>74</td>
<td>2344</td>
<td>132500</td>
<td>12054.39</td>
<td>11935.90</td>
<td>16568.84</td>
</tr>
<tr>
<td>MultihopOscilloscope</td>
<td>83</td>
<td>2376</td>
<td>133500</td>
<td>12069.20</td>
<td>16952.36</td>
<td>19176.07</td>
</tr>
<tr>
<td>MultihopOscilloscopeLqi</td>
<td>81</td>
<td>2176</td>
<td>133500</td>
<td>11935.90</td>
<td>15668.84</td>
<td>17732.80</td>
</tr>
<tr>
<td>Null</td>
<td>7</td>
<td>2008</td>
<td>68500</td>
<td>12961.71</td>
<td>22703.51</td>
<td>25442.00</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>64</td>
<td>2360</td>
<td>133500</td>
<td>12151.52</td>
<td>18173.07</td>
<td>22703.51</td>
</tr>
<tr>
<td>Powerup</td>
<td>10</td>
<td>2344</td>
<td>132500</td>
<td>23903.20</td>
<td>40923.93</td>
<td>40923.93</td>
</tr>
<tr>
<td>RadioCountToLeds</td>
<td>53</td>
<td>2368</td>
<td>133500</td>
<td>12207.81</td>
<td>18173.95</td>
<td>22703.51</td>
</tr>
<tr>
<td>RadioSenseToLeds</td>
<td>64</td>
<td>2368</td>
<td>133500</td>
<td>18173.95</td>
<td>18173.95</td>
<td>22703.51</td>
</tr>
<tr>
<td>Sense</td>
<td>29</td>
<td>2344</td>
<td>132500</td>
<td>13819.45</td>
<td>25442.00</td>
<td>25442.00</td>
</tr>
</tbody>
</table>

Table 8
Configuration Source File AST Sizes

There is a high degree of variability in AST size, as indicated by the large standard deviations. We therefore again consider both the mean and median values in our size estimates. In MultihopOscilloscope, the application with the largest number of configurations, the mean-based estimated memory consumed is 1001743.6 bytes (approximately 978kB). The median-based estimated memory consumed is 685248 bytes (approximately 669kB). We believe that this is an acceptable amount of memory to dedicate to the representation of all of the configurations within an application.

Module Space Requirements. Table 9 summarizes the space requirements associated with ASTs corresponding to nesC modules. The minimum size ranges from 1736 bytes to 6968 bytes. The maximum size ranges from 71500 bytes to 389500 bytes. The applications do not share common minimum sizes, suggesting that the smallest modules are not 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, in Powerup, the minimum size of 2808 bytes corresponds to the application-level module PowerupC, a module that uses 2 interfaces and implements 1 event that makes 1 function call. The maximum sizes do, however, indicate that the applications share a set of large core TinyOS modules. For example, the maximum size of 295500 bytes, shared by 4 of the applications, corresponds to $TOSDIR/chips/msp430/adc12/Msp430Adc12-ImplP.nc, a module that provides 5 interfaces, uses 15 interfaces, contains 1 enumeration with 9 elements, 6 state variables, and 16 non-empty functions.

Again, there is significant size variation, and we again consider both the mean and the median values in our total consumption estimates. In MultihopOscilloscope, the application with the largest number of modules, the mean-
Table 9
Module Source File AST Sizes

The median-based estimated memory consumed is 5100124.32 bytes (approximately 5MB). We believe this range represents an acceptable amount of memory to dedicate to the representation of all application modules.

Header File Space Requirements. Table 10 summarizes the space requirements associated with ASTs corresponding to nesC header files. The minimum size ranges from 680 bytes to 2992 bytes. The maximum size is consistently 1271500 bytes. As with other file types, the minimum and maximum values are shared across several programs, indicating shared TinyOS headers. 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 1271500 bytes, shared by all of 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 application relies on relatively few header files, and a single large file dominates the mean file size. As before, we consider both the mean and median values in our liberal size estimates. In MultihopOscilloscope, the application with the largest number of header files, the mean-based estimated memory consumed is 2308359.9 bytes (approximately 2MB). The median-based estimated memory consumed is 199500 bytes (approximately 194kB). Even using these liberal estimates, we believe this is an acceptable amount of memory to dedicate to the representation of all header files within an application.

Finally, Figure 7 compares the median AST sizes of each file type across all applications. ASTs corresponding to modules are typically the largest. This is not surprising; after all, the vast majority of application-level logic is im-

<table>
<thead>
<tr>
<th>Application</th>
<th>Files</th>
<th>Min</th>
<th>Max</th>
<th>Median</th>
<th>Mean</th>
<th>Stddev</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseStation</td>
<td>50</td>
<td>6968</td>
<td>256500</td>
<td>34000.00</td>
<td>60854.16</td>
<td>59198.91</td>
</tr>
<tr>
<td>Blink</td>
<td>21</td>
<td>5616</td>
<td>71500</td>
<td>28500.00</td>
<td>29719.24</td>
<td>18735.46</td>
</tr>
<tr>
<td>MViz</td>
<td>63</td>
<td>2816</td>
<td>389500</td>
<td>33500.00</td>
<td>67630.29</td>
<td>78173.27</td>
</tr>
<tr>
<td>MultihopOscilloscope</td>
<td>72</td>
<td>2848</td>
<td>37000.00</td>
<td>30000.00</td>
<td>70835.06</td>
<td>79563.37</td>
</tr>
<tr>
<td>MultihopOscilloscopeLqi</td>
<td>72</td>
<td>4832</td>
<td>35000.00</td>
<td>63997.22</td>
<td>64831.98</td>
<td>64831.98</td>
</tr>
<tr>
<td>Null</td>
<td>11</td>
<td>1736</td>
<td>27500</td>
<td>22289.82</td>
<td>21939.52</td>
<td>21939.52</td>
</tr>
<tr>
<td>Oscilloscope</td>
<td>55</td>
<td>4808</td>
<td>34500.00</td>
<td>58563.05</td>
<td>62995.55</td>
<td>62995.55</td>
</tr>
<tr>
<td>Powerup</td>
<td>14</td>
<td>2808</td>
<td>30000</td>
<td>29483.43</td>
<td>19969.64</td>
<td>19969.64</td>
</tr>
<tr>
<td>RadioCountToLeds</td>
<td>47</td>
<td>5640</td>
<td>29500</td>
<td>53206.10</td>
<td>53206.10</td>
<td>53206.10</td>
</tr>
<tr>
<td>RadioSenseToLeds</td>
<td>55</td>
<td>4816</td>
<td>34500</td>
<td>58072.87</td>
<td>61772.53</td>
<td>61772.53</td>
</tr>
<tr>
<td>Sense</td>
<td>30</td>
<td>4800</td>
<td>31000</td>
<td>45580.13</td>
<td>56955.85</td>
<td>56955.85</td>
</tr>
</tbody>
</table>
implemented in the form of module functions. 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 not representative of normal nesC applications, as it consists of very few source files.
6 Related Work

The difficulties associated with developing flexible 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 synchronous and asynchronous events, tasks, component wirings, and fanning of function calls and returns.

Object-level. Object-level program analysis and instrumentation tools target the compiler-generated binary images of programs. This post-compilation process is independent of the source language and compiler and does not require developers to recompile the target application — the original source is not needed. Many object-level tools have been presented in the literature for various hardware and software platforms. Tools for binary executables include ATOM [46], EEL [29], PatchWrx [7], Etch [40], Dyninst [6], Pin [33], LOPI [24], and others. Tools for Java bytecode include BCEL [10], JOIE [9], SERP [56], and SOOT [52]. 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 [37] 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. The framework supports the construction of a wide range of specialized analysis and instrumentation tools, the most popular of which is Memcheck [42]. 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 wireless sensor systems; its on-the-fly instrumentation approach makes it too heavyweight. Also, Valgrind supports only the Linux operating system, and only Intel- and PowerPC-class architectures, making it inapplicable to TinyOS applications running on the microcontrollers found in embedded wireless devices. Our source-based approach, however, 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 [5] is a framework that 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 that enable developers to interact with the systems under analysis (and/or instrumentation). The first is an 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). While the event-based approach is efficient for analysis activities, it does not easily support instrumentation. The second API is tree-based. This API exposes the Java class files as ASTs and provides a set of visitors for traversing and modifying those trees. Unlike our approach, where the nodes of the ASTs represent high-level language constructs, the nodes within ASM ASTs represent bytecode instructions and require developers to be familiar with the structure of that bytecode. Also unlike our approach, where the API provides methods for simplifying AST traversal, modification, and generation, all such activities are handled using visitor-based approaches. Finally, although bytecode manipulation frameworks, such as ASM, are useful for devices designed to execute Java bytecode directly, most wireless sensor systems are currently designed for the nesC/TinyOS platform.

Intermediate-level. Some programming languages, such as C++, are notoriously difficult to parse correctly [13]. In response to 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 g4re tool chain [27], for example, 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 enables users to access abstract semantic graphs (ASGs) that represent individual compilation units. The API enables users to iterate over the elements of each graph, or, like ASM, to access the elements using a visitor-based approach. The g4re 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, g4re provides information down to only the declaration level; expressions are not included. Also, g4re supports only analysis tasks — it is not capable of performing program instrumentation or source regeneration. Finally, while useful for analyzing C++ applications, g4re cannot be applied to nesC programs.

Source-level. Source-level program analysis and instrumentation tools target the source code of a program to be analyzed or instrumented. This analysis and instrumentation can be done by either instrumenting an existing compiler to collect the desired information, or by developing a custom parser for the language under analysis. The gccXfront [19] 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 source XML, graphically navigate the tree-like structure of the source XML, view the XSLT stylesheet used to transform the XML into a more easily readable form, and view XML documents after the XSLT transformations are applied. Unlike our approach, which provides an API that enables users to analyze a target application, gccXfront produces XML files that contain program information. Other tools are necessary to read those XML files and to perform the analysis activities. Also unlike our approach, gccXfront provides no means of performing program instrumentation. Finally, while gccXfront is useful for analyzing C, C++, and Java programs, it cannot be applied to nesC applications.

Compilers supporting aspect-oriented programming (AOP), such as AspectJ [25] and AspectC++ [44], may be considered source-level instrumentation tools. In AOP, concerns that cross-cut modules are factored out 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 [34], as well as to instrument programs with code to check temporal invariants [16]. Unlike our approach, which enables users to instrument a 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 applications, AOP can be used for only instrumentation.

More closely related to our work are tools that implement parsers for the language under analysis. One such tool is Columbus [13], 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 visualization. Columbus does not, however, support program instrumentation, nor is it readily adaptable to the nesC programming language.

We conclude by emphasizing two points. First, none of the prior work in this area has focused on analysis and instrumentation activities involving nesC (or nesC-like languages). Second, nesC is linguistically unique; existing analysis and instrumentation solutions simply do not apply. Hence, in this regard, our toolkit is the first of its kind.
7 Conclusion

Automated software engineering methods are integral to the development and maintenance of desktop- and enterprise-class systems. The tools that implement these methods are typically developed using libraries that provide common analysis and instrumentation services, and expose those services in a manner that simplifies and expedites tool construction. While many such toolkits exist for languages such as C, C++, and Java, there has been no prior work focused on languages tailored for sensor network construction — a burgeoning area of academic research and commercial development. To this end, we have developed a static analysis and instrumentation toolkit for nesC, the defacto standard for wireless sensor network development. We described the design and implementation of the toolkit and presented two use-case scenarios that demonstrate its utility. We additionally presented a detailed analysis of the toolkit’s runtime performance characteristics and memory usage, illustrating that the tool is appropriate for a wide range of software engineering applications. Our hope is that the toolkit will serve as a catalyst for extending software engineering research to this new domain, and that the resulting methods and tools will improve the development and maintenance of nesC-based systems.

Acknowledgments

The work presented in this article was funded in part by grants from the National Science Foundation (CNS-0745846, CNS-0520222). The authors gratefully acknowledge the NSF for its support.

References


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


