High Confidence TinyOS

John Regehr Phil Levis

An appropriate starting point

We believe that TinyOS and the nesC programming language have several properties that make them an ideal starting point for new directions in high confidence cyber-physical systems research. First, nesC has first-class language support for a clean component model. The current version of TinyOS, 2.0.1, is aggressively componentized, building upon five years of design experience with TinyOS 1.0 and 1.1. Unlike traditional systems software, TinyOS is built from large numbers of small components whose individual verification and validation (V&V) is tractable. For example, the core platform-independent system components have an average of 17 statements per component (the largest has 105). Second, TinyOS is small enough (total of around 28 KLOC) that we are not limited to totally automated formal methods. Also, as an alternative to unsound methods and heroic analyses (the current state of the art for analyzing larger systems such as Linux) we can co-design the language, the OS, and their V&V methods in parallel. For example, in recent work we found that an apparently minor design decision in a core interface—a return value by pointer parameter—introduced a significant roadblock in program analysis. The simple solution, which will be part of TinyOS 2.1, is to change the interface slightly in order to make the OS more checkable. Third, nesC strongly promotes static resource allocation, greatly simplifying V&V by avoiding the need to reason about, for example, worst-case heap behavior. Similarly, nesC's static program composition eliminates almost all of the run-time linking code that hinders static analysis of languages such as C++. Finally, nesC is a C dialect and can exploit the full power of C for low-level and performance-sensitive codes.

First steps towards high confidence

Our work to date has enabled analysis and transformation of TinyOS applications to ensure type safety, freedom from stack overflow, freedom from data races (our checker improves upon nesC's race checker by following pointers), freedom from interrupt overload, and absence of some kinds of interface usage errors. Through our participation in the TinyOS Alliance, we are currently working to incorporate these checks into the standard TinyOS distribution. Additionally, the challenges of developing reliable sensor network software have led us to explore ways to create more representative simulation environments in which to validate network protocols. Using approaches we have developed, network deployers can survey an environment, measure its wireless characteristics using low-cost sensor nodes and then recreate a representative environment in simulation.[1]

[1]Citations omitted due to space constraints. Papers are available at `http://www.cs.utah.edu/~regehr` and `http://csl.stanford.edu/~pal`.

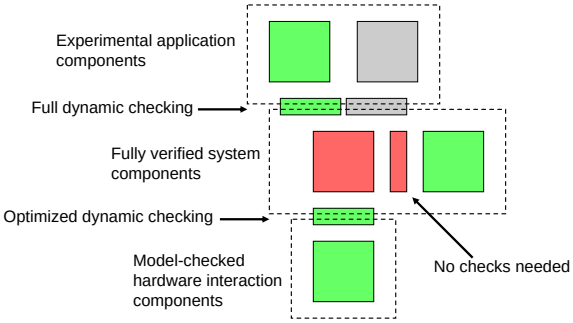

Figure 1: Verification and validation strategies can be mixed, as shown in this example, as long as component data integrity has been (otherwise) assured. Boxes represent components and rectangles are interfaces.

Moving forward: Mixed verification methods

We propose *multipurpose contracts*: specifications of the correct behavior of nesC interfaces (there are currently 60 "public" interfaces) designed to support diverse V&V methods:

- Automated unit testing using contracts as environment generators. In this scenario, a call into the component under test assumes the call's preconditions and asserts its postconditions, whereas a call out of the component asserts the call's preconditions and assumes its postconditions.

- Automated execution-driven testing of device drivers using symbolic methods to discover values for hardware registers—including interrupt masks—that result in good test coverage, in order to search for executions that violate contracts.

- Heavyweight verification of components and subsystems. Verification cannot be automated but it can be facilitated using advanced tools and by using assume-guarantee reasoning, exploiting TinyOS 2.0's heavy use of a relatively small number of simple, narrow interfaces. We have proven TinyOS's queue ADT correct in terms of an algebraic queue specification, and we are making progress towards verification of other core data structures. Although we certainly do not expect all of TinyOS to be verifiable, we do expect to be able to verify some of its important subsystems.

Importantly, V&V strategies can be mixed within individual applications. Figure 1 illustrates an experimental application that contains new—and almost certainly buggy—code. This application can and should reuse middleware that has been fully verified. Values flowing from new to verified code require full dynamic checking, whereas interfaces between verified components require no checking: they cannot go wrong. When components that have been heavily checked but not actually verified are reused, runtime checking can be optimized by omitting checks that correspond to behaviors that have been statically ruled out. As components are validated and verified, the boundaries between these different regions will shift. An incremental approach is important because it supports rapid and exploratory initial development, followed by successive application of increasingly heavyweight V&V techniques as software moves towards deployment.

In order to safely mix lightweight and heavyweight V&V techniques, the integrity of each component's data must be assured—even when some parts of the system are wholly untrusted. We believe that taken together, type safety, stack safety, and concurrency safety are sufficient to guarantee component integrity. Right now, we can check all of these properties for unmodified nesC code.

What's new?

Many tools and techniques exist for V&V of embedded programs written in C. Relative to this existing work we expect to make contributions in the following areas.

Contract language innovations: We plan to develop a contract language based on first order logic that contains a well-defined subset that can be translated into efficient executable checks. Specific novel features of our contract language will support: (1) formalization and checking of pointer protocols that are particular to sensor network software, such as TinyOS's buffer-swap idiom; (2) specification of properties of collections of device registers such as those used to access hardware timers, interrupt controllers, and packet-based radio interfaces; (3) specification of *semantically connected* interfaces that require checks that are stronger than those that apply to individual interfaces. For example, in TinyOS 2.0 it is common for access to a resource such as a sensor to be mediated by an arbiter component. The arbitration interface and the resource's interface are semantically connected in the sense that it is illegal for a client of the resource to use it without first being granted access by the arbiter.

Tool integration innovations: To make effective use of existing V&V tools for C code, we will need to faithfully translate nesC language features such as components, generic types, network types, parameterized interfaces, interrupt handlers, fan-in and fan-out, tasks, and atomic blocks. To compile programs that use mixed verification techniques, new algorithms will be needed, for example to insert the minimum number of dynamic checks needed to ensure data integrity for all components.

System design innovations: We will feed back lessons learned into the design of future iterations of TinyOS. For example, we expect that our work on checking pointer protocols will lead to revision of poorly-structured pointer idioms that cannot be easily formalized. Similarly, it is not, at present, completely clear what kinds of reentrant function calls a correct TinyOS component must be prepared to handle. Our goal is to develop an appropriate collection of rules about reentrancy and enforce them in future versions of the system.

John Regehr
50 S. Central Campus Dr., Room 3190
Salt Lake City, UT 84112-9205
+1 801 581 4280

Dr. Regehr is an assistant professor in the School of Computing at the University of Utah.

Phil Levis
358 Gates Hall
Stanford University
Stanford, CA 94305-9030
+1 650 725 9046

Dr. Levis is an assistant professor in the Department of Computer Science at Stanford University.