The nesC Language: A Holistic Approach to Networked Embedded Systems

David Gay, Phil Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler

IRB-TR-02-019

November, 2002

DISCLAIMER: THIS DOCUMENT IS PROVIDED TO YOU "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE. INTEL AND THE AUTHORS OF THIS DOCUMENT DISCLAIM ALL LIABILITY, INCLUDING LIABILITY FOR INFRINGEMENT OF ANY PROPRIETARY RIGHTS, RELATING TO USE OR IMPLEMENTATION OF INFORMATION IN THIS DOCUMENT. THE PROVISION OF THIS DOCUMENT TO YOU DOES NOT PROVIDE YOU WITH ANY LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS

Copyright 2002, Intel Corporation, All rights reserved.

The *nesC* Language: A Holistic Approach to Networked Embedded Systems

David Gay,[*] Phil Levis,[+] Robert von Behren,[+] Matt Welsh,[*] Eric Brewer[+] and David Culler[+*]

[+] *University of California at Berkeley,* [*] *Intel Research, Berkeley*
http://nescc.sourceforge.net/

Abstract

We present *nesC*, a programming language for networked embedded systems, such as sensor network "motes," which represent a new design space for application developers. Sensor networks consist of (potentially) thousands of tiny, low-power "motes," each of which execute concurrent, reactive programs that must operate with severe memory and power constraints.

nesC's contribution is to support the special needs of this domain by exposing a programming model that incorporates event-driven execution, a flexible concurrency model, and component-oriented application design. Restrictions on the programming model allow the nesC compiler to perform whole-program analyses, including data-race detection (which improves reliability) and aggressive function inlining (which reduces resource consumption).

nesC has been used to implement TinyOS, a small operating system for sensor networks, as well as several significant sensor applications. nesC and TinyOS have been adopted by a large number of sensor network research groups, and our experience and evaluation of the language shows that it is effective at supporting the complex, concurrent programming style demanded by this new class of deeply networked systems.

1. INTRODUCTION

Advances in networking and integration have enabled small, flexible, low-cost nodes that interact with their environment through sensors, actuators and communication. Single-chip systems are now emerging that integrate a low-power CPU and memory, radio or optical communication, and substantial MEMs-based on-chip sensors; these nodes are colloquially referred to as "motes" or "smart dust" [49]. Target costs (for single-chip motes) are less than 10 cents per unit, which enables networks with potentially tens of thousands of motes. Target power consumption means that motes can last years with low-bandwidth communication, or even be battery-free when fueled by ambient power (e.g., heat from the environment).

In this paper, we present nesC, a systems programming language for networked embedded systems such as motes. nesC supports a programming model that integrates reactivity to the environment, concurrency, and communication. By performing whole-program optimizations and compile-time data race detection, nesC simplifies application development, reduces code size, and eliminates many sources of potential bugs.

A key focus of nesC is holistic system design. Mote applications are deeply tied to hardware, and each mote runs a single application at a time. This approach yields three important properties. First, all resources are known statically. Second, rather than employing a general-purpose OS, applications are built from a suite of reusable system components coupled with application-specific code. Third, the hardware/software boundary varies depending on the application and hardware platform; it is important to design for flexible decomposition.

There are a number of unique challenges that nesC must address:

Driven by interaction with environment: Unlike traditional computers, motes are used for data collection and control of the local environment, rather than general-purpose computation. This focus leads to two observations. First, motes are fundamentally event-driven, reacting to changes in the environment (message arrival, sensor acquisition) rather than driven by interactive or batch processing. Second, event arrival and data processing are concurrent activities, demanding an approach to concurrency management that addresses potential bugs such as race conditions.

Limited resources: Motes have very limited physical resources, due to the goals of small size, low cost, and low power consumption. We do not expect new technology to remove these limitations: the benefits of Moore's Law will be applied to reduce size and cost, rather than increase capability. Although our current motes are measured in square centimeters, a version is in fabrication that measures less than 5 mm^2.

Reliability: Although we expect individual motes to fail due to hardware issues, we must enable very long-lived applications. For example, environmental monitoring applications need to collect data without human interaction for months at a time. An important goal is to reduce run-time errors, since there is no real recovery mechanism in the field except for automatic reboot.

Soft real-time requirements: Although there are some tasks that are time critical, such as radio management or sensor polling, we do not focus on hard real-time guarantees. Our experience so far indicates that timing constraints are easily met by having complete control over the application and OS, as well as by limiting utilization. One of the few timing-critical aspects in sensor networks is radio communication; however, given the fundamental unreliability of the radio link, it is not necessary to meet hard deadlines in this domain.

Although nesC is a synthesis of many existing language concepts targeted at the above problems, it provides three broad contributions. First, nesC defines a component model that supports event-driven systems: the model provides bidirectional interfaces to simplify event flow, supports a flexible hardware/software boundary, and admits efficient implementation that avoids virtual functions and dynamic component creation. Second, nesC defines a simple but expressive concurrency model coupled with extensive compile-time analysis: the nesC compiler detects most data races at compile time. This combination allows applications to exhibit highly concurrent behavior with very limited resources. Third, nesC provides

Mote Type	WeC	rene2	rene2	dot	mica
Date	9/99	10/00	6/01	8/01	2/02
Microcontroller					
Type	AT90LS8535		ATMega163		ATMega103
Prog. mem. (KB)	8		16		128
RAM (KB)	0.5		1		4
Default Power source					
Size	CR2450	2xAA		CR2032	2xAA
Capacity (mAh)	575	2850		225	2850
Communication					
Radio	RFM TR1000				
Rate (Kbps)	10	10	10	10	10/40
Modulation type	OOK				OOK/ASK

Table 1: *The family of TinyOS motes.*

a unique balance between accurate program analysis to improve reliability and reduce code size, and expressive power for building real applications. In addition to static data race detection, the nesC compiler performs static component instantiation, whole-program inlining, and dead-code elimination. We prohibit many features that hinder static analysis, including function pointers and dynamic memory allocation, but are capable of supporting complex applications and a substantial user community.

nesC is used as the programming language for TinyOS [16], a small operating system for sensor network applications that is in use by more than 100 research groups worldwide. Several significant sensor network applications have been implemented in nesC, including *TinyDB* [29], a sensor network query processing engine, and *Maté* [28], a small virtual machine that allows rapid reprogramming of sensor networks.

Section 2 presents background material on sensor networks and introduces *Surge*, a sample nesC application used as a running example throughout the paper. Section 3 presents the nesC design; Section 4 summarizes our experience with nesC and evaluates the effectiveness of data-race detection and inlining. We conclude with a survey of related work (Section 5) and a discussion of nesC and its future directions (Section 6).

2. BACKGROUND

Wireless sensor networks are composed of large numbers of tiny resource-limited devices called "motes." A first application of these networks is data collection in uncontrolled environments, such as nature reserves [30] or seismically threatened structures [25]. Four key features have emerged in networks we have deployed: interaction with the local environment though sensors, communication via a wireless network, lifetime requirements of months to a year, and physical inaccessibility.

Table 1 presents several generations of motes designed at UC Berkeley. Although very resource constrained, motes must be highly reactive and participate in complex distributed algorithms, such as data aggregation [19, 29] or spatial localization [50]. This combination of requirements makes traditional operating systems and programming models inappropriate for sensor networks. Mote hardware evolves rapidly: Table 1 covers five platforms in three years, with different sensors and varying levels of hardware support for operations such as radio-based messaging. A sensor network operating system and programming language must make it easy for applications to adapt to these changes.

2.1 TinyOS

TinyOS [16] is an operating system specifically designed for network embedded systems. TinyOS has a programming model tailored for event-driven applications as well as a very small footprint (the core OS requires 396 bytes of code and data memory, combined). Two of our motivations in designing nesC were to support TinyOS's programming model and to reimplement TinyOS in the new language. TinyOS has several important features that influenced nesC's design: a component-based architecture, a simple event-based concurrency model, and split-phase operations.

Component-based architecture: TinyOS provides a set of reusable system *components*. An application connects components using a *wiring specification* that is independent of component implementations; each application customizes the set of components it uses. Although most OS components are software modules, some are thin wrappers around hardware; the distinction is invisible to the developer. Decomposing different OS services into separate components allows unused services to be excluded from the application. We present nesC's support for components in Sections 3.1 and 3.2.

Tasks and event based concurrency: There are two sources of concurrency in TinyOS: *tasks* and *events*. Tasks are a deferred computation mechanism. They run to completion and do not preempt each other. Components can *post* tasks; the post operation immediately returns, deferring the computation until the scheduler executes the task later. Components can use tasks when timing requirements are not strict; this includes nearly all operations except low-level communication. To ensure low task execution latency, individual tasks must be short; lengthy operations should be spread across multiple tasks. The lifetime requirements of sensor networks prohibit heavy computation, keeping the system reactive.

Events also run to completion, but may preempt the execution of a task or another event. Events signify either completion of a split-phase operation (discussed below) or an event from the environment (e.g. message reception or time passing). TinyOS execution is ultimately driven by events representing hardware interrupts. We discuss the refinement and inclusion of TinyOS's concurrency model into nesC in Sections 3.3 and 3.4.

Split-phase operations: Because tasks execute non-preemptively, TinyOS has no blocking operations. All long-latency operations are *split-phase*: operation request and completion are separate functions. *Commands* are typically requests to execute an operation. If the operation is split-phase, the command returns immediately and completion will be signaled with an event; non-split-phase operations (e.g. toggle an LED) do not have completion events. A typical example of a split-phase operation is a packet send: a component may invoke the `send` command to initiate the transmission of a radio message, and the communication component signals the `sendDone` event when transmission has completed. Each component implements one half of the split-phase operation and calls the other; the wiring connects both commands and events across component boundaries. We discuss how nesC captures split-phase operations in Section 3.1.

Resource contention is typically handled through explicit rejection of concurrent requests. In the example above, if the communication component cannot handle multiple concurrent `send` operations, it signals an error when a concurrent `send` is attempted. Alternately, the communication component may queue the request for future processing.

The simple concurrency model of TinyOS allows high concurrency with low overhead, in contrast to a thread-based concurrency model in which thread stacks consume precious memory while blocking on a contended service. However, as in any concurrent system, concurrency and non-determinism can be the source of complex bugs, including deadlock (e.g. the Mars Rover [22]) and

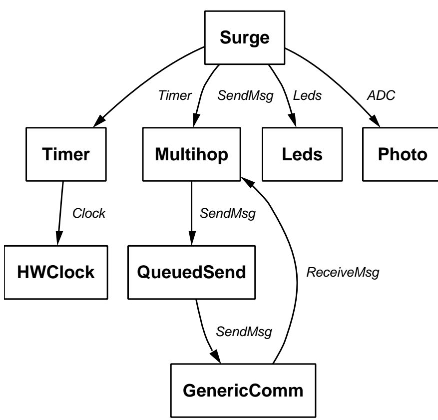


Figure 1: *Simplified view of the Surge application. Nodes represent components, and edges represent interface wiring. Edges are labeled with the corresponding interface name.*

data races (e.g. the Therac-25 [27]). One of the primary benefits of nesC is helping the programmer use concurrency safely by ensuring the absence of (most) data races. This is discussed in Section 3.3.

2.2 Surge: A Sensor Network Application

A common application of sensor networks is to periodically sample a sensor (e.g., light or temperature) and report readings to a base station, which is typically a node with a wired network connection and power source. As a running example throughout this paper, we present *Surge*, a simple application that performs periodic sensor sampling and uses ad-hoc multi-hop routing over the wireless network to deliver samples to the base station. Surge is intentionally simple and does not perform advanced functions such as in-network data aggregation [19, 29].

Surge motes organize themselves into a spanning tree rooted at the base station. Each mote maintains the address of its *parent* and its *depth* in the tree, advertising its depth in each radio message (either sensor sample or forwarded message) that it transmits. A node selects an initial parent by listening to messages and choosing the node with the smallest depth; to seed the creation of the spanning tree, the base station periodically broadcasts beacon messages with depth 0. Nodes estimate parent link quality; when the link quality falls below some threshold, nodes select a new parent from their neighbor set based on link quality estimates and depth.

Once a second, each mote samples its light sensor and sends the sample to its parent. Parents acknowledge received packets. Surge uses the acknowledgments to provide a reliable transport layer; parent link quality is calculated as the fraction of transmitted messages that are acknowledged. When a node receives a message from another node, it forwards the message to its parent. Sensor samples are collected at the base station where they can be analyzed or visualized.

Figure 1 shows the components of Surge and the interfaces by which they are wired. This example shows several important advantages of components. The Surge application only needs to include the parts of TinyOS that it needs, i.e., system boot code (`Main`), the timer (`Timer`), a sensor (`Photo`), access to the LEDs (`Leds`), and multi-hop message routing (`Multihop`). Secondly, the application code – the `Surge` component – explicitly specifies its environmental dependencies in terms of interfaces. `Surge` requires a timer (`Timer` interface), sensor (`ADC` interface), LEDs (`Leds` interface), and communication (`Send` interface). The code is therefore independent of the particular sensor hardware used. For example, `ADC` could easily be wired to a temperature sensor rather than a light sensor, as long as the two provide the same interface.

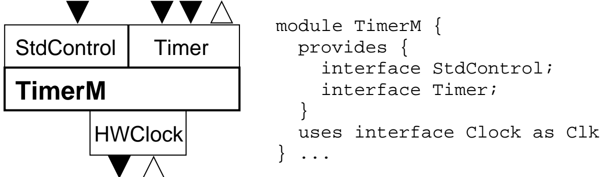


Figure 2: *Specification and graphical depiction of the* `TimerM` *component.*

3. *nesC* DESIGN

In this section we discuss the primary concepts in nesC's design. First, nesC applications are built out of *components* with well-defined, bidirectional *interfaces*. Second, nesC defines a concurrency model, based on tasks and events, and detects data races at compile time.

A few basic principles underlie nesC's design:

nesC is an extension of C: C [23] produces efficient code for all the target microcontrollers that are likely to be used in sensor networks. C provides all the low-level features necessary for accessing hardware, and interaction with existing C code is simplified. Last but not least, many programmers are familiar with C.

C does have significant disadvantages: it provides little help in writing safe code or in structuring applications. nesC addresses safety through reduced expressive power and structure through components. None of the new features in nesC are tied to C: the same ideas could be added to other imperative programming languages such as Modula-2 [52].

Whole-program analysis: nesC programs are subject to whole program analysis (for safety) and optimization (for performance). Therefore we do not consider separate compilation in nesC's design. The limited code space on motes makes this approach tractable.

nesC is a "static language": There is no dynamic memory allocation and the call-graph is fully known at compile-time. These restrictions make whole program analysis and optimization significantly simpler and more accurate. They sound more onerous than they are in practice: nesC's component model and parameterized interfaces eliminate many needs for dynamic memory allocation and dynamic dispatch. We have, so far, implemented one optimization and one analysis: a simple whole-program inliner and a data-race detector. Details are given in Section 4.

nesC supports and reflects TinyOS's design: nesC is based around the concept of components, and directly supports TinyOS's event-based concurrency model. Additionally, nesC explicitly addresses the issue of concurrent access to shared data (Section 3.3).

3.1 Component Specification

nesC applications are built by writing and assembling *components*. A component *provides* and *uses* interfaces. These interfaces are the only point of access to the component. An interface generally models some service (e.g., sending a message) and is specified by an *interface type*. Figure 2 shows the `TimerM` component, part of the TinyOS timer service, that provides the `StdControl` and `Timer` interfaces and uses a `Clock` interface (all shown in Figure 3). `TimerM` provides the logic that maps from a hardware clock (`Clock`) into TinyOS's timer abstraction (`Timer`).

```
interface StdControl {
  command result_t init();
}

interface Timer {
  command result_t start(char type, uint32_t interval);
  command result_t stop();
  event result_t fired();
}

interface Clock {
  command result_t setRate(char interval, char scale);
  event result_t fire();
}

interface Send {
  command result_t send(TOS_Msg *msg, uint16_t length);
  event result_t sendDone(TOS_Msg *msg, result_t success);
}

interface ADC {
  command result_t getData();
  event result_t dataReady(uint16_t data);
}
```

Figure 3: *Some interface types.*

Interfaces in nesC are *bidirectional*: they contain *commands* and *events*. A command is a function that is implemented by the providers of an interface, an event is a function that is implemented by its users. For instance, the `Timer` interface (Figure 3) defines `start` and `stop` commands and a `fired` event. In Figure 2 provided interfaces are shown above the `TimerM` component and used interfaces are below; downward-pointing arrows depict commands and upward-pointing arrows depict events. Although this same interaction between the timer and its client could have been provided via two separate interfaces (one for `start` and `stop`, and one for `fired`), grouping these commands and events in the same interface makes the specification much clearer and helps prevent bugs when wiring components together. Split-phase operations are cleanly modeled by placing the command request and event response in the same interface. Figure 3 shows two examples of this. The `Send` interface has the `send` command and `sendDone` event of the split-phased packet send (Section 2.1). The `ADC` interface is similarly used to model split-phase sensor value reads.

The separation of interface type definitions from their use in components promotes the definition of standard interfaces, making components more reusable and flexible. A component can provide and use the same interface type (e.g., when interposing a component between a client and service), or provide the same interface multiple times. In these cases, the component must give each *interface instance* a separate name using the `as` notation shown for `Clk` in Figure 2.

Components are also a clean way to abstract the hardware/software boundary. For instance, on one sensor board, the temperature sensor (accessed via a component named `Temp`) is mostly in hardware; `Temp` is a thin layer of software accessing on-chip hardware registers. On another it is accessed over an I^2C bus; `Temp` is a component implemented as a number of interacting components including a generic I^2C access component.

3.2 Component Implementation

There are two types of components in nesC: *modules* and *configurations*. Modules provide application code, implementing one or more interfaces. Configurations are used to wire other components together, connecting interfaces used by components to interfaces provided by others. Every nesC application is described by a *top-level configuration* that wires together the components used.

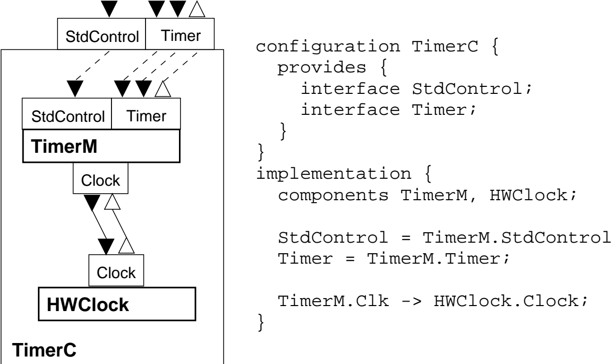
```
module SurgeM {
  provides interface StdControl;
  uses interface ADC;
  uses interface Timer;
  uses interface Send;
}
implementation {
  uint16_t sensorReading;

  command result_t StdControl.init() {
    return call Timer.start(TIMER_REPEAT, 1000);
  }

  event result_t Timer.fired() {
    call ADC.getData();
    return SUCCESS;
  }

  event result_t ADC.dataReady(uint16_t data) {
    sensorReading = data;
    ... send message with data in it ...
    return SUCCESS;
  }
  ...
}
```

Figure 4: *Simplified excerpt from* `SurgeM`.




```
configuration TimerC {
  provides {
    interface StdControl;
    interface Timer;
  }
}
implementation {
  components TimerM, HWClock;

  StdControl = TimerM.StdControl;
  Timer = TimerM.Timer;

  TimerM.Clk -> HWClock.Clock;
}
```

Figure 5: *TinyOS's timer service: the* `TimerC` *configuration.*

The body of a module is written in C-like code, with straightforward extensions. A command or event f in an interface i is named $i.f$. A command *call* is like a regular function call prefixed with the keyword `call`, similarly an event `signal` is like a function call prefixed by `signal`. The definition of a commands or event named $i.f$ is prefixed with `command` or `event`. We require these annotations to improve code clarity. Figure 4 is a simplified excerpt from `SurgeM`, which is part of the Surge application. It defines the `StdControl.init` command, called at boot-time, and two of the events handled by Surge: the firing of the timer (`Timer.fired`) and sensor data acquisition (`ADC.dataReady`). The code calls the `Timer.start` command to setup periodic timer events and the `ADC.getData` command to request a new sensor sample. Modules have private state, in this example the `sensorReading` variable.

`TimerC`, the TinyOS timer service, is implemented as a configuration, shown in Figure 5. `TimerC` is built by wiring the two subcomponents given by the `components` declaration: `TimerM` (from Figure 2) and `HWClock` (access to the on-chip clock). It maps its `StdControl` and `Timer` interfaces to those of `TimerM` (`StdControl = TimerM.StdControl`, `Timer = TimerM.Timer`) and connects the hardware clock interface used by `TimerM` to that provided by `HWClock` (`TimerM.Clk -> HWClock.Clock`). Figure 6 shows a more elaborate example: the toplevel configuration for the Surge application.

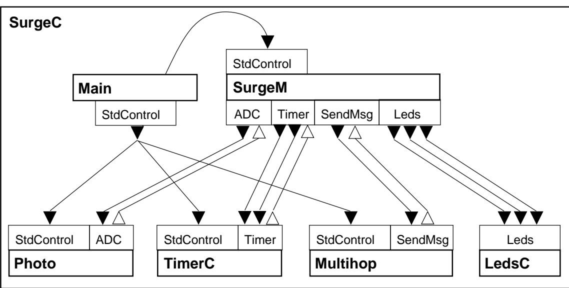


Figure 6: *The* `SurgeC` *configuration: A top-level configuration.*

An interface of a component may be wired zero, one or more times. As a result, an arbitrary number of command call expressions may be wired to a single command implementation ("fan-in"), and a single command call expression may be connected to an arbitrary number of command implementations ("fan-out"). For instance, Figure 6 shows that calls to `StdControl.init` in `Main` are connected to four different implementations (in `SurgeM`, `Photo`, `TimerC` and `Multihop`). nesC allows a fan-out degree of zero (no wires) if the module implementer provides a "default" implementation for the unwired command. Fan-out degrees greater than one are allowed as long as the return type of the command is associated with a function for combining the results of all the calls. In the case of `StdControl.init`, the result type `result_t` (Figure 3) represents success or failure. Its combining function provides logical-and-like behavior, thus the result of the call to `StdControl.init` in `Main` is success if all four implementations succeed. The analogous situations for event signal expressions is handled identically.

The explicit wiring of components via interfaces, combined with the removal of function pointer types[1], makes the control-flow between components explicit. Module variables are private and,, as a design style in TinyOS, we discourage sharing of data amongst components. Taken together, this makes it much easier to write correct components and understand their behavior when wired in an application.

Most components in TinyOS represent services (such as the timer) or pieces of hardware (such as the LEDs) and therefore exist only in a single instance. However, it is sometimes useful to create several instances of a component. In nesC, this is achieved by declaring an *abstract component* with optional parameters; abstract components are created at compile-time in configurations. For instance, the `QueuedSend` component used by the multi-hop communication layer `Multihop` (Figure 1) is an abstract component that takes a maximum retransmit count parameter:

```
abstract module QueuedSend(int maxAttempts) { ... }

configuration Multihop {
  provides interface Send;
}
implementation {
  components MultihopM, QueuedSend(10) as newQueue, ... ;

  Send = MultihopM.Send;
  MultihopM.QueuedSendMsg -> newQueue.Send;
  ...
}
```

[1] At this point our implementation issues a warning when function pointers are used.

3.3 Concurrency and Atomicity

Data races occur due to concurrent updates to shared state. In order to prevent them, a compiler must 1) understand the concurrency model, and 2) determine the target of every update. In this section we present the concurrency model and the key invariant that the compiler must enforce to avoid data races. We achieve tractable target analysis by reducing the expressive power of the language and performing alias analysis. In particular, nesC has no dynamic memory allocation and no function pointers.

In TinyOS, code runs either asynchronously in response to an interrupt, or in a synchronously scheduled task. To facilitate the detection of race conditions, we distinguish synchronous and asynchronous code:

> **Asynchronous Code (AC):** code that is reachable from at least one interrupt handler.
>
> **Synchronous Code (SC):** code that is only reachable from tasks.

The run-to-completion rule and sequential execution of tasks lead immediately to a key invariant:

> *Invariant: Synchronous Code is atomic with respect to other Synchronous Code.*

By "atomic," we mean that any shared state between the two will by updated atomically. This essentially provides atomicity by default for tasks. Code that includes split-phase operations, which by definition must include (at least) two tasks, is not atomic as a whole, although each half is atomic. We discuss larger units of atomicity in Section 6.

Although non-preemption allows us to avoid races amongst tasks, there are still potential races between SC and AC, as well as AC and AC. We claim that:

> *Claim 1: Any update to shared state from AC is a potential race condition.*
>
> *Claim 2: Any update to shared state from SC that is also updated from AC is a potential race condition.*

To reinstate atomicity in these cases, the programmer has two options: either to convert all of the sharing code to tasks (SC only), or to use *atomic sections* to update the shared state. An atomic section is a small code sequence that nesC ensures will run atomically. We present the syntax and implementation issues for atomic sections in Section 3.4. We require that any update to shared state that is a potential race condition based on the claims above, must occur within an atomic section. This gives us our basic invariant:

> **Race-Free Invariant***: Any update to shared state is either not a potential race condition (SC only), or occurs within an atomic section.*

In nesC, we enforce this invariant at compile time, which avoids most race conditions. Note that this invariant only guarantees that individual accesses are race-free; incorrect use of atomic sections can still lead to race conditions.

3.4 Concurrency in nesC

Concurrency is central to nesC components: events (and commands) may be signaled directly or indirectly by an interrupt, which makes them asynchronous code. To handle this concurrency, nesC provides two tools: atomic sections and tasks. Figure 7 illustrates

```
module SurgeM { ... }
implementation {
  bool busy;
  norace uint16_t sensorReading;

  event result_t Timer.fired() {
    bool localBusy;
    atomic {
      localBusy = busy;
      busy = TRUE;
    }
    if (!localBusy)
      call ADC.getData();
    return SUCCESS;
  }

  task void sendData() { // send sensorReading
    adcPacket.data = sensorReading;
    call Send.send(&adcPacket, sizeof adcPacket.data);
    return SUCCESS;
  }

  event result_t ADC.dataReady(uint16_t data) {
    sensorReading = data;
    post sendData();
    return SUCCESS;
  }
  ...
}
```

Figure 7: *Concurrency and atomicity in* `SurgeM`*. Changes from Figure 4 are highlighted.*

their use, showing the core logic of the Surge application. Here, `Timer.fired` and `ADC.dataReady` are asynchronous code.

The `Timer.fired` event is signaled periodically. If Surge is not already busy, `ADC.getData` is called to get a new sensor value. As `busy` is accessed in asynchronous code, its use is protected by an `atomic` statement that performs a test-and-set on `busy`. When the sensor value is available, the `ADC.dataReady` event is signaled. Sending the message with the sensor reading is not a time-critical operation, and the TinyOS communication layer is not designed to be executed as asynchronous code. Therefore, `SurgeM` posts a task, `sendData`, which sends the sensor reading message. Posted tasks are executed by the TinyOS scheduler when the processor is idle.

We currently implement `atomic` by disabling and enabling interrupts, which has very low overhead (a few cycles). However, leaving interrupts disabled for a long period delays interrupt handling, which makes the system less responsive. To minimize this effect, atomic statements are not allowed to call commands or signal events, either directly or in a called function. This confines the code executed by an `atomic` statement to a single module, making it possible for the module implementer to bound atomic statement execution time.

As discussed in Section 3.3, `atomic` prevents concurrent access to the shared data accessed in the statement. Given atomic sections, we define the key rule that enforces our race-free invariant:

> If a variable x is accessed by AC, then any access of x outside of an atomic statement is a compile-time error.

The compiler reveals the specific conflict in addition to signaling the error. To remove the error, the programmer must either add an atomic section, or move the offending code into a task.

There are some cases in which there is a *potential* race condition on a variable that the programmer knows is not an actual race condition, for instance `sensorReading` in Figure 7. The declaration of `sensorReading` includes the `norace` qualifier to suppress errors about this particular false positive. This avoids uselessly protecting all accesses to `sensorReading` with `atomic` statements.

```
module GenericComm { // id is the Active Message ID
  provides interface Send[uint8_t id];
  provides interface Receive[uint8_t id];
} implementation {
  TOS_Msg *msg;

  command result_t
    Send.send[uint8_t id](uint8_t length, TOS_Msg *data)
  { data->amId = id; msg = data; ... }

  void sendComplete(TOS_Msg *packet) {
    signal Send.sendDone[msg->amId](msg, SUCCESS);
  }
  ...
}

configuration Multihop { ... }
implementation {
  components QueuedSend(10) as newQueue, GenericComm,
  ...
  newQueue.RealSend -> GenericComm.Send[42];
}
```

Figure 8: *Active Messages using Parameterized Interfaces.*

Section 4.2 shows that we can effectively detect a large number of data races.

3.5 Parameterized Interfaces

Parameterized interfaces are nesC's mechanism for introducing runtime command and event dispatch within a first order language. A component declares an interface with a parameter list which creates a separate interface for each tuple of parameter values. Parameterized interfaces are used to model Active Messages [47] in TinyOS: in Active Messages, packets contain a numeric identifier that specifies which event handler should be executed. Figure 8 shows a very simplified definition of the communication component (`GenericComm`). Wires to a parameterized interface must specify a specific interface with a compile-time constant: the multihop message routing uses Active Message 42 for all its communication, hence wires to `GenericComm.Send[42]` in Figure 8.

In a module, the implemented commands and events of a parameterized interface receive extra parameters specifying the selected interface (see `Send.send` in the `ActiveMessages` module of Figure 8) and select a specific interface when invoking a command or event in a parameterized interface (see `sendComplete`). This last construction translates into a runtime dispatch to one of the functions connected to `Send.sendDone` by the application's configurations. However the set of possible dispatch targets is explicitly specified by the program's configurations. An additional benefit is that no RAM is needed to store the active message dispatch table.

4. EVALUATION

In this section we evaluate nesC's component model, concurrency model, and whole-program inlining with respect to a set of representative TinyOS applications, including Surge, TinyDB and Maté. We have implemented a compiler for nesC that generates a single C source file for a whole application, resolving all interface connections to direct function calls. In the results below, this C file is compiled with gcc 3.1.1 for the Atmel ATmega 103, an 8-bit RISC microprocessor used on Mica mote (Table 1).

4.1 Component Model

Anecdotally, nesC's component model has been invaluable for event-driven sensor applications. The success of the component model is shown by the way in which components are used in the TinyOS code. The core TinyOS source consists of 186 components,

Component	Type	Data-race variables
RandomLFSR	System	1
UARTM	System	1
AMStandard	System	2
AMPromiscious	System	2
BAPBaseM	Application	2
ChirpM	Application	2
MicaHighSpeedRadioM	System	2
TestTimerM	Application	2
ChannelMonC	System	3
NoCrcPacket	System	3
OscilloscopeM	Application	3
QueuedSend	System	3
SurgeM	Application	3
SenseLightToLogM	Application	3
TestTemp	Application	3
MultihopM	System	10
eepromM	System	17
TinyAlloc	System	18
IdentC	Application	23
Total		103

Figure 9: *Component locations of race condition variables.*

of which 121 are code modules and 65 are configurations. The number of modules per application ranges from 20 to 69, with an average of 35. Modules are generally quite small — on average only 144 lines of code. This small size indicates the expressive power of nesC's components, as programmers have not needed to break the model by producing large monolithic components, which are more difficult to analyze and more error prone.

nesC's bidirectional interfaces are an excellent fit for event-driven systems, since they provide a clean syntax for grouping related computation in the presence of split-phase and asynchronous operations. As evidence of this, bidirectional interfaces are pervasive in TinyOS: of the 186 components in the TinyOS core, 68% utilize at least one bidirectional interface.

4.2 Concurrency

nesC's component model makes it simple to express the complex concurrent actions in sensor network applications. Our sample applications each have an average of 17 tasks and 75 events, each of which represents a potentially concurrent activity. Moreover, in our sample applications an average of 43% of the code (measured in bytes) is reachable from an interrupt context, demonstrating a high degree of concurrency.

Our implementation of race detection uses a simple type-based alias analysis to detect which variables are accessed by asynchronous code. We report errors if any of these variables are accessed outside atomic sections.

Initially, nesC included neither an explicit `atomic` statement nor the analysis to detect potential race conditions; TinyOS and its applications had many data races. Once race detection was implemented, we analyzed every application in the TinyOS source tree, finding 156 variables that potentially had a race condition. Of these, 53 were false positives (discussed below) and 103 were genuine race conditions, a frequency of about six per thousand code statements. We fixed each of these bugs by moving code into tasks or by using `atomic` statements. We tested each TinyOS application and verified that the presence of atomic sections has not interfered with correct operation.

Figure 9 shows the locations of data races in the TinyOS tree. Half of the races existed in system-level components used by many applications, while the other half were application-specific. `MultihopM`, `eepromM`, and `TinyAlloc` had a disproportionate number of races due to the amount of internal state they maintain through complex concurrent operations.

```
/* Contains a race: */
if (state == IDLE) {
  state = SENDING;
  count++;
  // send a packet
}
```

```
/* Fixed version: */
uint8_t oldState;
atomic {
  oldState = state;
  if (state == IDLE) {
    state = SENDING;
  }
}
if (oldState == IDLE) {
  count++;
  // send a packet
}
```

Figure 10: *Fixing a race condition in a state transition.*

The finite state machine style of decomposition in TinyOS led to the most common form of bug, a non-atomic state transition. State transitions are typically implemented using a read-modify-write of the state variable, which must be atomic. A canonical example of this race is shown in Figure 10, along with the fix.

The original versions of the communication, `TinyAlloc` and EEPROM components contained large numbers of variable accesses in asynchronous code. Rather than using large atomic sections, which might decrease overall responsiveness, we promoted many of the offending functions to synchronous code, by posting a few additional tasks.

False positives fell into three major categories: state-based guards, buffer swaps, and causal relationships. The first class, state-based guards, occurred when access to a module variable is serialized at run time by a state variable. The above state transition example illustrates this; in this function, the variable `count` is safe due to the monitor created by `state`.

TinyOS's communication primitives use a buffer-swapping policy for memory management. When a network packet is received, the radio component passes a buffer to the application; the application returns a separate buffer to the component for the next receive event. This raises problems with alias analysis; although only one component has a reference to a given buffer at any time, components swap them back and forth. Although alias analysis would conclude that both components could concurrently read and write the same buffer, swapping ensures that only one component has a reference to it at any time. To resolve this issue, we annotated message buffers with the `norace` qualifier.

The last class of false positives, causal relationships, comes from the split-phase operation of TinyOS components. The command/event pair of a split-phase operation might share a variable, but their causal relationship means that the event will not fire while the command is executing. The event could fire during the command only if another TinyOS component accessed the underlying hardware; although this would violate the TinyOS programming model, nesC does not enforce this limitation.

4.3 Optimization

As with data-race detection, nesC exploits the restrictions of the component model to perform static analysis and optimization. The nesC compiler uses the application call-graph to eliminate unreachable code and to inline small functions. These optimizations greatly reduce memory footprint, a key savings for embedded systems. Figure 11 shows a breakdown of the code and data size for each component in the Surge application. The **TinyOS** row represents the core TinyOS initialization code and task scheduler, which fits into 396 bytes. **C Runtime** represents necessary runtime routines, including floating-point libraries (currently used by multihop routing). Dead code elimination trimmed off an initial 9% of the Surge program code, and inlining yielded an additional 16%

Component	Code size		Data size
(Sizes in bytes)	*inlined*	*noninlined*	
Runtime			
TinyOS	364	666	32
C Runtime	1152	1162	13
`RealMain`	–	72	0
Application components			
`SurgeM`	80	240	44
Multi-hop communication			
`AMPromiscuous`	456	654	9
`MultihopM`	2646	2884	223
`NoCRCPacket`	370	484	50
`QueuedSend`	786	852	461
Radio stack			
`ChannelMonC`	454	486	9
`CrcFilter`	–	34	0
`MicaHighSpeedRadioM`	1162	1250	61
`PotM`	50	82	1
`RadioTimingC`	42	56	0
`SecDedEncoding`	662	684	3
`SpiByteFifoC`	344	438	2
Sensor acquisition			
`ADCM`	156	260	2
`PhotoTempM`	248	360	2
Miscellaneous			
`NoLeds`	–	18	0
`RandomLFSR`	134	134	6
`TimerM`	1826	1734	118
Hardware presentation			
`HPLADCC`	214	268	11
`HPLClock`	74	134	0
`HPLInit`	–	10	0
`HPLInterrupt`	–	22	0
`HPLPotC`	–	66	0
`HPLSlavePinC`	–	28	0
`HPLUARTM`	160	212	0
`LedsC`	–	164	1
`SlavePinM`	80	124	1
`UARTM`	78	136	1
Totals	11538	13714	1050

Figure 11: *Breakdown of code and data size by component for the* `Surge` *application. A '–' in the* inlined *column indicates that the corresponding component was entirely inlined.*

savings. Note that inlining increases code size slightly for only one component – `TimerM` – but reduces or eliminates many others, leading to an overall reduction in footprint. Inlining provides only a small performance increase (less than a 1% improvement in CPU utilization for our sample applications) – our CPU intensive code (mostly low level parts of the radio stack) is not dominated by function calls.

5. RELATED WORK

The module systems of languages such as Modula-2 (plus descendants) [12, 52, 53] and Ada [20] explicitly import and export interfaces. However these systems are less flexible than nesC, as there is no explicit binding of interfaces: an exported interface I is automatically linked to all importers of I. Standard ML's module system offers similar functionality to nesC, except that circular "component" (`structure`) assemblies cannot be expressed. The nesC module system is very close to Mesa's [35], and (coincidentally) uses essentially the same terminology: *modules* contain executable code, *configurations* connect components (configurations or modules) by binding their interfaces, whose *interface types* must match.

Giotto [13], Esterel [4], Lustre [11], Signal [3] and E-FRP [48] are languages that target embedded, hard real-time, control systems. They explicitly model the concept of (input) events and (output) control signals, and offer much stronger time guarantees than nesC. However, they are not general purpose programming languages in which one would implement, e.g., a multi-hop radio stack.

The VHDL [18] hardware description language is based on assembling components ("architecture") with well-defined interfaces. Architectures are often specified as independent processes; architectures that are built out of components always encapsulate the inner components. In contrast, nesC configurations do not encapsulate the components they use, and concurrency crosses component boundaries. VHDL's model matches that of actual hardware, while nesC's is inspired by hardware. Another language targeted to generating hardware is SAFL [38], a first-order functional language. To allow hardware generation, SAFL has no recursion and static data allocation.

Distributed systems [14, 21, 34, 40, 41, 45] and software engineering [2] often model systems as interacting sets of components. These components are specified by the interfaces they provide or use. However, the focus is very different from nesC: components are large-scale (e.g., a database), dynamically loaded and/or linked, and possibly accessed remotely.

ArchJava [1] has bidirectional interfaces (*ports*). There are no interface types (port connections match methods by name and signature), fan-out is limited to methods with no result, and dispatch of port methods is dynamic. C# [32] and BCOOPL [5] support events in classes and interfaces. However the two directions are asymmetric: registration of an event handler with an event producer is dynamic.

The languages above, and languages commonly used for programming embedded systems (such as C, Ada, Forth), do not offer the set of features desired in nesC: an interrupt-based concurrency model, low-level hardware access, component-based programming and static concurrency checking.

A number of operating systems have explored the use of component architectures. The Flux OSKit [10] is a collection of components for building operating systems, but provided components target workstation-like machines. Flux was subsequently reimplemented with the help of Knit [43], a language for constructing and linking components implemented in C. Knit's component model, based on that of units [9], is similar to nesC in that components provide and use interfaces and that new components can be assembled out of existing ones. Unlike nesC, Knit lacks bidirectional interfaces and data race detection. THINK [8] takes the Flux model one step further by allowing explicit modeling of calls between components. This allows clean linking of THINK components across protection domains or networks. THINK does not employ whole-program optimization and relies on dynamic dispatch. We do not believe this model, with its associated runtime cost, is appropriate for network embedded systems. Other component-oriented systems include Click [36], Scout [37], and the x-kernel [17]. These systems are more specialized than nesC and do not support whole-program optimization (apart from several optimizations in Click [24]) or bidirectional interfaces.

Traditional real-time and embedded operating systems, such as VxWorks [51], QNX [42], and WinCE [33], differ from TinyOS in a number of respects. These systems are generally much larger and provide greater functionality than TinyOS, and are intended for larger-scale embedded systems. We refer the reader to [16] for a thorough discussion of the differences between these systems.

There have been a few attempts at static detection of race conditions. ESC [7] is in between a type checker and a program verifier; it has been used to verify user-supplied associations between locks and variables, and also to enforce ordering constraints on lock acquisition. Sun's LockLint [46] statically checks for inconsistent use of locks or lock ordering. As expected, these tools have trouble

with first-class functions and aliasing, and tend to report a subset of the errors with false positives as well. They also focus on locking rather than atomicity; we chose the latter to enable more freedom of implementation, which is particularly important for interrupts. The next section covers the use of monitors in Mesa.

There are also tools for dynamic detection of races. Eraser [44] detects unprotected shared variables using a modified binary. "On the fly" race detectors [31, 39] serialize all accesses to a variable to verify serializability. These approaches only catch errors that actually occur during the test run. In addition, dynamic approaches are less appealing for motes due to their number, resource limitations, and UI constraints. All of these race detection systems, including nesC, validate individual variable accesses. They cannot detect a read-modify-write through a temporary variable in which the read and write occur in distinct atomic sections.

6. DISCUSSION AND FUTURE WORK

The nesC language is well-suited to the unique challenges of programming networked embedded systems. nesC was originally designed to express the concepts embodied in TinyOS, and by reimplementing the operating system in nesC, those concepts were refined. nesC's component model supports holistic system design by making it easy to assemble applications that include only the necessary OS support. The component model also allows alternate implementations and a flexible hardware/software boundary. nesC's concurrency model allows us to write highly concurrent programs on a platform with very limited resources; the use of bidirectional interfaces and atomic statements permit a tight integration of concurrency and component-oriented design. Careful restrictions on the programming model, including the lack of dynamic allocation and explicit specification of an application's call-graph, facilitate whole-program analyses and optimizations. Aggressive inlining reduces memory footprint, and static data-race detection allows the developer to identify and fix concurrency bugs.

The nesC design opens up several key areas for future work. These broadly fall into the areas of concurrency support, enhancements to language features, and application to domains other than networked embedded systems.

Concurrency support

The nesC concurrency model provides short atomic actions, which can be used to build higher-level synchronization mechanisms such as semaphores, condition variables, atomic queues, and locks. Some of these mechanisms imply blocking, but there is nothing in the language *per se* that prevents support for blocking: we would need to prohibit blocking calls in atomic sections as well as treat blocking calls as yield points for task scheduling.

Our current implementation of atomic sections, which works well for embedded systems, is to disable interrupts. This is acceptable in part because we prevent blocking and limit the length of atomic sections. It also depends on the assumption of a uniprocessor and on the lack of virtual memory, since a page fault should not occur within an atomic section. However, these assumptions can be relaxed by considering alternate implementations of atomic sections, for example, using non-blocking synchronization primitives [15].

The use of monitors in Mesa [26] is the most directly comparable concurrency model to ours. In Mesa, the authors considered and rejected atomicity based on non-preemption for several reasons, including the desire to support multiprocessors and virtual memory. Also, non-preemption alone does not handle interrupts; we use atomic sections to handle asynchronous updates to shared state. Finally, Mesa was unable to prevent calls within atomic sections from yielding the processor. This is not an issue for nesC.

Language enhancements

There are a number of idioms common in TinyOS that are not well expressed in nesC. Multi-client services with per-client state are not well-supported. For example, consider a general timer service where each client wishes to receive timer events at a different frequency. Abstract components can be used for this purpose, although they are currently limited in that the internal state of each instance is private to that instance. We currently use parameterized interfaces to implement such multi-client services, where the parameter corresponds to the "client number." We are not wholly satisfied with this approach, and we plan to investigate a better mechanism in the future.

Split-phase operations provide high concurrency with low overhead, but are difficult to program; reintroducing the convenience of a threaded model would greatly simplify programming. By automatically transforming blocking operations into split-phase calls, we could preserve expressive lightweight concurrency without forcing the programmer to manually build continuations within components (as they do now). As it stands, many components are written as small finite state machines; atomic state transitions result in replicated control flow, separating state transitions from their corresponding actions. A future direction for nesC is to provide explicit support for FSM-style decomposition that simplifies component design and allows properties of FSM behavior to be statically verified.

Application to other platforms

We believe that nesC is not limited to the domain of embedded systems. nesC's component-oriented structure, focus on concurrency, and bidirectional interfaces are valuable concepts in programming larger systems, such as enterprise-class applications and Internet services. To effectively support this broader class of applications, several extensions to nesC are needed. First, nesC's compile-time analyses would need to be extended to handle dynamic memory and component allocation, as well as patterns such as message buffer swap. The static checking of software protocols in Vault [6] may provide an approach to solving these problems. nesC's concurrency model should be extended to admit multiprocessors, blocking operations, and a more general notion of threads, as discussed above. Such an approach would lead to a rich set of concurrency primitives specifically tailored for component-oriented programming of large-scale systems.

REFERENCES

[1] J. Aldrich, C. Chambers, and D. Notkin. Architectural Reasoning in ArchJava. In *European Conference on Object Oriented Programming (ECOOP)*, June 2002.

[2] F. Bachmann, L. Bass, C. Buhrman, S. Cornella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition. Technical Report CMU/SEI-2000-TR-008, Carnegie Mellon Software Engineering Institute, May 2000.

[3] A. Benveniste, P. L. Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, Sept. 1991.

[4] F. Boussinot and R. de Simone. The ESTEREL Language. *Proceedings of the IEEE*, 79(9):1293–1304, Sept. 1991.

[5] H. de Bruin. BCOOPL: A Langage for Controlling Component Interactions. *Journal of Supercomputing*, 2002. Accepted for publication.

[6] R. Deline and M. Fahndrich. Enforcing High-level Protocols in Low-Level Software. In *Proceedings of the ACM SIGPLAN '01*

Conference on Programming Language Design and Implementation, pages 59–69, June 2001.

[7] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report #159, Palo Alto, USA, 1998.

[8] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. THINK: A Software Framework for Component-based Operating System Kernels. In *Proceedings of Usenix Annual Technical Conference*, June 2002.

[9] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[10] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the Symposium on Operating Systems Principles*, pages 38–51, 1997.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[12] S. P. Harbison. *Modula-3*. Prentice Hall, 1991.

[13] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Embedded Control Systems Development with Giotto. In *Proceedings of the ACM Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 64–72, June 2001.

[14] A. Herbert. An ANSA Overview. *IEEE Network*, 8(1):18–23, 1994.

[15] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[16] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000. TinyOS is available at http://webs.cs.berkeley.edu.

[17] N. C. Hutchinson and L. L. Peterson. Design of the *x*-kernel. In *Proceedings of SIGCOMM '88*, pages 65–75, Aug. 1988.

[18] IEEE Standard 1076-2002. *VHDL Language Reference Manual*.

[19] C. Intanagonwiwat, D. Estrin, R. Govindan, and J. Heidemann. Impact of network density on data aggregation in wireless sensor networks. In *Proceedings of International Conference on Computing Systems (ICDCS)*, July 2002.

[20] International Organisation for Standardization. *Ada 95 Reference Manual*, Jan. 1995.

[21] ISO/IEC International Standard 10746-3. *ODP Reference Model: Architecture*, 1995.

[22] M. Jones. What really happened on mars rover pathfinder. *The Risks Digest*, 19(49).

[23] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988.

[24] E. Kohler, B. Chen, M. F. Kaashoek, R. Morris, and M. Poletto. Programming language techniques for modular router configurations. Technical Report MIT-LCS-TR-812, MIT Laboratory for Computer Science, Aug. 2000.

[25] Lab Notes: Research from the College of Engineering, UC Berkeley. *Smart buildings admit their faults.*, 2001. http://coe.berkeley.edu/labnotes/1101.smartbuildings.html.

[26] B. W. Lampson and D. D. Redell. Experience with processes and monitors in mesa. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, pages 43–44, 1979.

[27] N. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[28] P. Levis and D. Culler. Mate: a Virtual Machine for Tiny Networked Sensors. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2002.

[29] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proceedings of the ACM Symposium on Operating System Design and Implementation (OSDI)*, Dec. 2002.

[30] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless Sensor Networks for Habitat Monitoring. In *Proceedings of the ACM International Workshop on Wireless Sensor Networks and Applications*, Sept. 2002.

[31] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Proc. of Supercomputing '91*, pages 24–33, 1991.

[32] *Microsoft C# Language Specification*. Microsoft Press. ISBN 0-7356-1448-2.

[33] Microsoft. *Windows CE*. http://www.microsoft.com.

[34] *OLE2 Programmer's Reference, Volume One*. Microsoft Press, 1994.

[35] J. Mitchell. Mesa language manual. Technical Report CSL-79-3, Xerox PARC, 1979.

[36] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 217–231, 1999.

[37] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Operating Systems Design and Implementation*, pages 153–167, 1996.

[38] A. Mycroft and R. Sharp. A Statically Allocated Parallel Functional Language. In *Proceedings of the Internal Conference on Automata, Languages and Programming (ICALP)*, pages 37–48, 2000.

[39] R. H. B. Netzer. Race condition detection for debugging shared-memory parallel programs. Technical Report CS-TR-1991-1039, 1991.

[40] Object Management Group. *Common Object Request Broker Architecture*. Available at http://www.omg.org.

[41] Object Management Group. *CORBA Component Model (CCM) Specification*. Available at http://www.omg.org.

[42] QNX Software Systems, Ltd, Kanata, Ontatio, Canada. http://www.qnx.com.

[43] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide. Knit: Component composition for systems software. In *Proceedings of the 4th ACM Symposium on Operating System Design and Implementation*, pages 347–360, Oct. 2000.

[44] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[45] Sun Microsystems. *Enterprise Java Beans*. Available at http://java.sun.com/ejb.

[46] SunSoft. *Sun Workshop 5.0 Collection: Chapter 5, Lock Analysis Tool*, 2000.

[47] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the International Symposium on Computer Architecture*, pages 256–266, 1992.

[48] Z. Wan, W. Taha, and P. Hudak. Event-Driven FRP. In *Proceedings of the Internation Symposium on Principles of Declarative Languages (PADL)*, volume 2257, pages 155–172, 2001.

[49] B. Warneke, M. L. andl B. Liebowitz, and K. Pister. Smart dust: Communicating with a cubic-millimeter computer. *IEEE Computer Magazine*, pages 44–51, January 2001.

[50] K. Whitehouse and D. Culler. Calibration as Parameter Estimation in Sensor Networks. In *Proceedings of the ACM International Workshop on Wireless Sensor Networks and Applications*, Sept. 2002.

[51] Wind River Systems, Inc, Alameda, CA, USA. http://www.vxworks.com.

[52] N. Wirth. *Programming in Modula-2*. Springer Verlag, 1992.

[53] N. Wirth and M. Reiser. *Programming in Oberon - Steps Beyond Pascal and Modula*. Addison-Wesley, 1992. ISBN 0-201-56543-9.