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

[by David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler]

Presented by Mayank Agrawal

Outline

- About network Embedded Systems
- About TinyOS
- About nesC
- Working of nesC
- An Example
- Related and Future work

Network embedded systems

- Small, flexible, low-cost nodes that interact with their environment through sensors, actuators and communication
- Single-chip systems that integrate a low-power CPU and memory, communication and substantial MEMS-based on chip sensors
- Example: Wireless sensor network motes

Challenges Involved

- Motes are driven by interaction with environment
- They have limited resources
- Their applications require reliability
- Soft real-time requirements

What is nesC

- nesC is 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

About the Environment - TinyOS

- Component-based architecture – All TinyOS functionality is contained by a set of reusable components
- Tasks and event-based concurrency – Execution is event based, i.e. it is ultimately driven by hardware interrupts
- Split-phase operations – Mechanism required for long latency operations as TinyOS has no blocking operations

Principle of nesC Design

- nesC is an extension of C – Uses the necessary C features to deal with hardware
- Allows for Whole-program analysis – Separate compilation is not considered
- It is a "static language" – No dynamic memory allocation. Makes whole-program analysis easier
- Supports and reflects TinyOS design – Additionally it addresses the issue of concurrent access to shared data

Features of nesC

- A component model that supports event driven systems
- Expressive concurrency model coupled with extensive compile time analysis
- Balance between accurate program analysis and expressive power for building real applications

Components

Components in TinyOS represent services (such as the timer) or pieces of hardware (such as the LEDs). nesC applications are built by writing and assembling *components*

- A component *provides* and *uses* interfaces.

- Two types of components in nesC: *modules* and *configurations*

Interfaces

Interfaces are what provide the actual services to application. Each one has a number of functions associated with itself that are called by other interfaces that need its services

- Interfaces are the only point of access to the component
- Generally models some service and is specified by an *interface type*
- They are bidirectional: they contain *commands* and *events*, both of which are essentially functions

Commands and Events

These are the functions associated with the interfaces that implement the actual functionality of the interface

- A command *call* is like a regular function call prefixed with the keyword call – These are defined by a component when it is providing that particular interface
- Event *signal* is like a function call prefixed by signal - These are defined by components when they use a particular interface

Modules

Modules contain the implementation part of a component with respect to function definitions of all the functions associated with its interfaces

- Modules provide application code, implementing one or more interfaces
- They are written in C-like code, with straightforward extensions

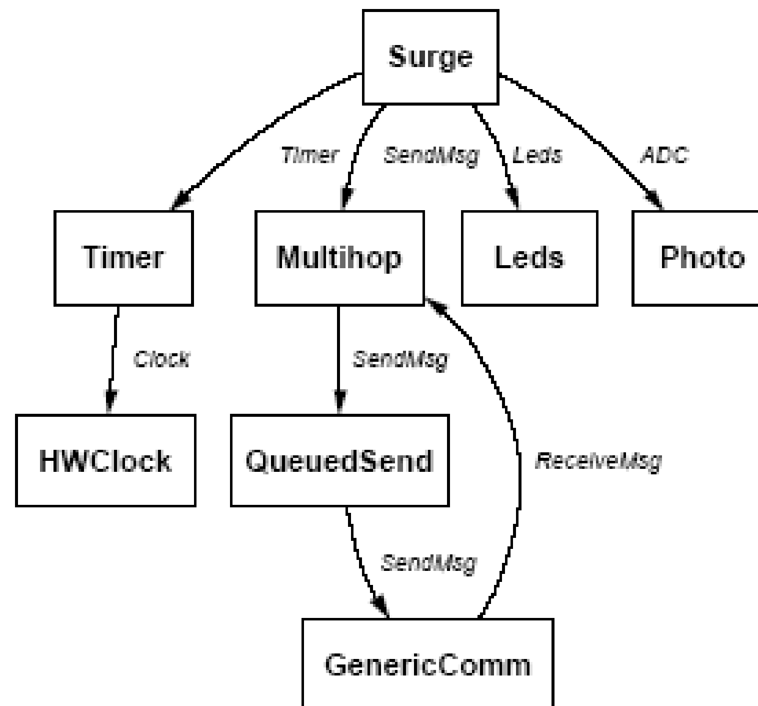
Configurations

The actual wiring together of interfaces is done in the implementation of a configuration. This is where instances of used components are to be declared.

- 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

Surge - An Example Application



Simplified view of the application and its components

The Surge Module

```
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;
  }
  ...
}
```

This clearly shows the definitions of the functions of its interfaces

Commands of provided interfaces defined

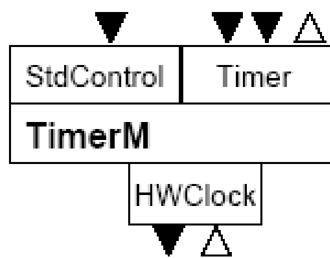
Events of used interfaces defined

Surge Interfaces

```
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);
}
```

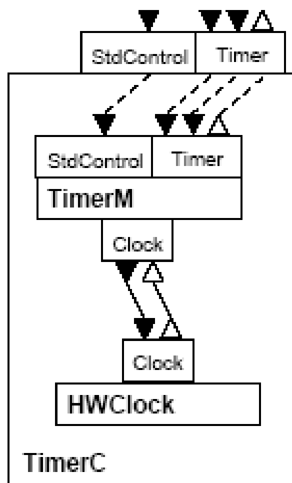
Shows what commands and events associated with all of Surge's interfaces

The Timer Component




```
module TimerM {
  provides {
    interface StdControl;
    interface Timer;
  }
  uses interface Clock as Clk;
} ...
```

Configuration and implementation of the Timer Component




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

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

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

Concurrency and Atomicity

- Concurrency is handled by two tools: atomic sections and tasks
- An atomic section is a small code sequence that nesC ensures will run atomically.

Optimizations Possible in nesC

- Reduction in memory footprint
- Reductions in execution overheads
- Overall reductions in code size – Inline small functions

Evaluation Results for some applications

Application	Modules	OS Modules (% of full OS)	Lines	OS Lines (% of full OS)
Surge	31	27 (25%)	2860	2160 (14%)
Maté	35	28 (25%)	4736	2524 (17%)
TinyDB	65	38 (35%)	11681	4160 (28%)

This table shows the effectiveness of the Component Model

Application	Task count	Event count	% interrupt code
Surge	7	221	64%
Maté	13	317	41%
TinyDB	18	722	57%

This table shows the effectiveness of the Concurrency Model

Future Work

- Concurrency support
- Language enhancements
- Application to other platforms

References

- David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler, The nesC Language: A Holistic Approach to Networked Embedded Systems, Proc. of Programming Language Design and Implementation (PLDI), June 2003.

Questions ???