nesC: 1.1 Bumps and Future Directions

David Gay, Intel Research, Berkeley
(and the nesC and TinyOS teams)
Status

►►

A year ago: TinyOS 0.6, no nesC

Last October: nesC 1.0, TinyOS 1.0 released
  - much improved usability
  - interfaces

This July: nesC 1.1, TinyOS 1.1
  - nesC made aware of TinyOS concurrency model
    - data race detection
  - many small nesC improvements
  - new TinyOS features (platforms, components)

In the future:
  - abstract components, generic interfaces
  - improved configurations, modules
module SurgeM {
    provides interface Init;
    uses interface ADC;
    uses interface Timer;
    uses interface Send;
}
implementation {
    ...
}
nesC 1.1

- Concurrency model
  - data race detection

- Small language improvements
  - `uniqueCount`
  - combiners for multiply-wired commands, events

- Tool improvements
  - `debugging on the mote via JTAG`
  - `ncg`: nesC constant generator to complement mig
  - `%T (TinyOS path), %P (platform) in –I directives`
  - `Easier to add new platforms, sensor boards`
Concurrency model

- TinyOS concurrency model:
  - tasks
    - run-to-completion
    - atomic with respect to each other
  - events (interrupts)
    - run-to-completion
  - no blocking
    - split-phase operations

- nesC concurrency features:
  - async for commands, events executable asynchronously
  - atomic guarantees atomic execution of a statement
  - data race detection helps fix concurrency bugs
module SurgeM { ... }

implementation {
    bool busy;
    async event void Timer.fired() { // From interrupt
        // Avoid concurrent data collection attempts!
        if (!busy) { // Concurrent state access
            busy = TRUE; // Concurrent state access
            call ADC.getData();
        }
    }
    ...
}

Race condition example
Data race detection

Every concurrent state access is a potential race condition

Concurrent state access:
- If object O is accessed in a function reachable from an interrupt entry point, then all accesses to O are potential race conditions
- All concurrent state accesses must occur in atomic statements

Concurrent state access detection is straightforward:
- Call graph fully specified by configurations
- Interrupt entry points are known
- Data model is simple (variables only)
module SurgeM { ... }
implementation {
  bool busy;
  async event void Timer.fired() { // From interrupt
    // Avoid concurrent data collection attempts!
    bool localBusy;
    atomic {
      localBusy = busy;
      busy = TRUE;
    }
    if (!localBusy)
      call ADC.getData();
  }
}
Multiple-client support using parameterised interfaces (instances identified by integer)

- **users:**
  ```c
  App.Timer -> TimerC.Timer[unique("Timer")];
  ```

- **implementation:**
  ```c
  int state[NUM_CLIENTS];
  command Timer.start[int id](int rate) {
    ... state[id] ...
  }
  ```

- **issue:** how to know NUM_CLIENTS?

- **in nesC 1.1:**
  ```c
  int state[uniqueCount("Timer")];
  ...
Combiners

►► in nesC 1.0:
  - x is result of A or B

►► in nesC 1.1:
  - x is combination of A and B’s result

configuration App { }
implementation {
  components App, A, B;
  App.StdControl -> A.StdControl;
  App.StdControl -> B.StdControl;
}

interface StdControl {
  command result_t init();
}

In module App:
  x = call StdControl.init();
  // what is x?

typedef char result_t attribute((combine (rcombine)));
result_t rcombine(result_t x, result_t y) {
  if (x == FAIL) return x;
  return y;
}
ncg

- mig (nesC 1.0) generates Java types based on nesC types
- ncg (nesC 1.1) generates Java constants based on nesC constants

```
SimpleCmdMsg.h:
enum {
    LED_ON = 1,
    LED_OFF = 2,
    RADIO_QUIETER = 3,
    ...
}
```

```
java file:
public class SomeName {
    public static final byte LED_ON = 1;
    public static final byte LED_OFF = 2;
    public static final byte RADIO_QUIETER = 3;
}
```
TinyOS 1.1

►► Update to new concurrency features in nesC 1.1
  ▪ many data races fixed in core system components
  ▪ async specified for appropriate interfaces

►► New features
  ▪ mica2, mica2dot support
    ▶ radio stack, extra UART, hardware I²C
    ▶ Random side note: mica2dot is slower than mica2
  ▪ Secure networking (TinySec)
  ▪ Matchbox: a filing system for the flash chip
    ▶ Also faster direct flash access
  ▪ new GenericBase with framing
    ▶ also preserves CRC, destination
    ▶ old GenericBase will still be supported
∃nesC x such that x > 1.1

- Configuration improvements
  - Abstract components, *generic interfaces*
  - “Multi-client services”
  - *Wiring checks*
  - *Alternate implementations of core OS services*
  - Automatic wiring?

- Module improvements
  - *better data race detection*
  - *more compile-time bug detection...*
  - threadlets
Abstract components

► Allow components to be instantiated several times
  ▪ compile-time instantiation (in configurations)
► Instantiations specify arguments (including types)

abstract module MessageQueue(int size) {
  provides interface SendMsg;
  uses interface SendMsg as RealSend;
} implementation {
  TOS_MsgPtr queue[size];
  ...

In some configuration:
  components new MessageQueue(10) as MyQueue;
  App.SendMsg -> MyQueue.SendMsg;
  MyQueue.RealSend -> GenericComm.SendMsg[22];
“Multi-client services”

- unique, uniqueCount ugly and error-prone
  
  ```
  App.Timer -> TimerC.Timer[unique("Timer")];
  ... state[uniqueCount("Timer")];
  command Timer.start[int id](int rate) {
    ... state[id] ...
  }
  ```

- Goal: extend language to allow for simpler, cleaner expression of such services
Automatic wiring?

 ► Idea: move scheduler to a component

```java
interface Task {
    command void post();
    event void execute();
}

module App {

}

implementation {
    ... post task1(); ...

    task void task1() {
        ... } }
```
Automatic wiring?

► Idea: move scheduler to a component

```plaintext
interface Task {
    command void post();
    event void execute();
}
```

```plaintext
module App {
    uses interface Task as Task1;
    uses interface Task as Task2;
}
implementation {
    ... call Task1.post(); ...

    event void Task1.execute() {
        ...
    }
}
```

► Issue: need to select scheduler, wire in many components

► Need some kind of global/automatic wiring?
module SurgeM { ... }
implementation {
    event void Timer.fired() {
        call ADC.getData(); // Start data collection
    }
    TOS_Msg m;
    event void ADC.dataReady(int data) {
        m.data = data; // Save sensor data
        post sendMsg(); // Cannot send in interrupt
    }
    task void sendMsg() {
        call Send.send(&m); // Start sending message
        // Message not yet sent!
    }
    event void Send.sendDone(...) { ... } // Message now sent
}
Threadlets

module SurgeM { ... }
implementation {
    TOS_Msg m;
    event void Timer.fired() {
        call ADC.getData(); // Start data collection
        when ADC.dataReady(int data) {
            intask {
                m.data = data;
                call Send.send(&m); // Start sending message
                when Send.sendDone(...) {
                    // Message now sent
                    ...
                }
            }
        }
    }
}
Conclusion

- nesC and TinyOS have come a long way in a year
  - usable language, fully implemented
    - very few compiler bugs
  - interfaces
  - data race detection

- New concerns have arisen
  - Need even better ways to assemble application, OS
    - abstract components, automatic wiring, wiring checks
  - Need more support for application code
    - better data race detection
    - easier split-phase programming