



nesC Update

Eric Brewer

with help from David Culler,
David Gay, Phil Levis, Rob von
Behren, and Matt Welsh



Language Goals

- **Short term (NesC)**
 - Make it much easier to program single motes
 - Gain experience with the issues (please tell us!)
 - Build intermediate language for future work
- **Long term (macroprogramming)**
 - Write high-level programs for groups of motes
 - Deal with failure and uncertainty, varying numbers of motes
 - Abstract issues of time, location, neighbors
 - Provide implicit communication and data sharing
 - Enable low power and bandwidth efficiency
- **Apply outside of TinyOS (?)**



Completed Goals [from June]

- **Easier coding and better code size**
- **Definition of messages**
 - Define a message type and then create surrounding code
- **Big goal: race-free, deadlock-free programs (done!)**
- **Documentation Tool**



Principles

- **C Like**
- **Whole-program analysis**
 - Enforce global properties (e.g. no races)
 - Optimization
- **“static language”**
 - no malloc, no function pointers
- **Support TinyOS: components and events**



Abstract Components

Multiple instances of the same component:

```
components AMHandler(38) as commandHandler,  
          AMHandler(45) as dataHandler;
```

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

```
configuration Multihop {  
  provides interface Send;  
}
```

```
implementation {  
  components MultihopM, QueuedSend(10)  
    as newQueue, ... ;  
  Send = MultihopM.Send;  
  MultihopM.QueuedSendMsg -> newQueue.Send;  
  ...  
}
```



Multi-client services?

- **Example: timer service with different intervals for different clients**
- **How to manage the shared state of the service?**
 - **Abstract components:**
 - » **State is private**
 - » **Added a way to name other instances state...**
 - **Parameterized components:**
 - » **Parameter is client ID – have to statically allocate**
 - » **Still awkward...**
 - **Other ways??**



Optimization

- **Whole-program analysis**
- **Code size reduction:**
 - Dead code elimination = 9%
 - Inlining = 16%
 - 25% total
- **Performance:**
 - About the same
 - Inlining surprisingly not helpful



Data Races

- **Key idea: detect sharing, enforce atomicity**
- **Two kinds of contexts:**
 - Interrupts: run any time
 - Tasks: scheduled, run to completion
 - » Code is atomic relative to other tasks
- **Any code that is reachable from an interrupt is “interrupt context”**
- **Key invariant:**
 - “atomic section” for anything shared with an interrupt context
 - We detect context statically, then enforce the invariant.



Atomic Sections

```
atomic {  
    <Statement list>  
}
```

- **We ensure it will be atomic.**
- **Restrictions:**
 - No loops
 - No yields (or blocking calls in the future)
 - No commands/events
 - Calls OK, but callee must meet restrictions
- **Easy to use in practice!**

Example

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



Results

- **Tested on full TinyOS tree, plus applications**
 - 186 modules (121 modules, 65 configurations)
 - 20-69 modules/app, 35 average
 - 17 tasks, 75 events on average (per app)
 - » Lots of concurrency!
- **Found 156 races: 103 real (!)**
 - About 6 per 1000 lines of code
 - 53 false positives
- **Fixing races:**
 - Add atomic sections
 - Post tasks (move code to task context)



False Positives

- **Three Categories**
 - **State-based guards**
 - » **If (state = x) update foo**
 - » **State check prevents race**
 - Probably good to still use atomic
 - **Buffer swaps**
 - » **Parent/child alternate ownership**
 - Hidden protocol for sharing
 - » **Could remove this class with “pass”**
 - **Causal Relationships**
 - » **Conflicting code segments are both runnable at the same time**
 - » **E.g., task runs only after timer interrupt**



Tools

- **“norace” keyword**
 - Use to remove false positives
- **“interrupt” keyword**
 - Means that it is OK to execute command/event in interrupt context
 - Claim: people should know when code runs via interrupts
- **Compile-time errors**
 - Potential races
 - » Fix: atomic, post or norace
 - Unexpected interrupt context
 - » Fix: change to task, or add ‘interrupt’ keyword

Still can have races...

- **Basic pattern:**
 - Read in one atomic section, write in another

```
atomic {  
    Tmp = x;  
}  
  
...  
tmp = f(tmp);  
  
...  
atomic {  
    x = tmp;  
}
```



NesC Future Work

- **Simplify split-phase calls**
 - Blocking calls at the higher level?
 - At least simplify creation of the command/event pair
 - » Reduce false positives?
 - Manage state across split-phase calls

- **“pass” values (like message buffer)**
 - Pass by reference, but object **leaves** your scope
 - Implies passed thing is NOT shared
 - Simplifies capabilities, events, pipelines
 - Helps the “buffer swap” case
 - » Reduces false positives



NesC Future Work

- **Dynamic Linking?**
 - Enable larger programs, multi-mode motes
 - Load modules from EEPROM? from network?
- **Scheduling?**
 - There is one simple scheduler with no knobs...
- **Sleep modes/timing?**
- **Explicit support for state machines?**
 - Would reduce false positives
 - Lot of modules have state, this would simplify them...



Conclusions

Significant progress!

- **Better code, faster development**
- **Fewer errors**
 - Race detection
 - Wiring errors
 - Interfaces
- **Code size down 25%**
- **Message tool (mig) done**
- **Doc tool done**
- **Concurrency checking works**