Beyond Event Handlers: Programming Wireless Sensor Networks with Attributed State Machines

Oliver Kasten and Kay Römer IPSN 2005
Problem Statement

• Programming these things is "somewhat tricky."

• It would be nice if it were less tricky.

• Let’s work on that.
Difficulties

- Low-level programming is messy and error-prone
- Verifying program correctness /debugging is difficult
- Code maintenance/readability tends to be sacrificed for efficiency.
- Most embedded systems software design is rooted in the event-driven design paradigm by necessity, which is not necessarily the most “natural” way to solve problems.
The 0x40 Dollar Question

- How do we map an event-driven platform to useful high-level abstractions without compromising performance?
* nesC/TinyOS: Make event-handling one of the primary features of the language.

* TOSThreads, ¬Contiki, Mantis: Implement a threading library on top of event-driven model to hide the details of event-handling from applications.

* Maté, other VMs: Provide a set of high-level APIs to application programmers so they never touch event-handlers.

* Other approaches?
I Would Argue...

- Purely event-driven approach suffers because it doesn’t have a good narrative structure-- design and implementation diverge.

- Threaded approach suffers because it glosses over the reality of the platform: embedded systems by their nature, should be responsive to things that happen in their environment.

- VMs can be heavy and rigid, where lightness and flexibility is key. Plus, we’d like our abstractions to be purely logical and have them translate to machine code that is still highly efficient, which is quite a trick in VMs.
So how do we combine flexibility, responsiveness, and a good story?

(according to google image search)
How ‘bout them Attributed Finite State Machines?

- We sketch out TinyOS programs as FSMs when we design them.
- People have spent a lot of effort trying to derive FSMs from programs (Kothari et al, 2008).
- They map really nicely to the features of the platform.
- But there’s not much in the way of direct implementations.
Oh yeah, the paper.

- Presents the Object State Model (OSM).
- Ideas based on StateCharts (Harel, 1987) / UML flow charts
- Actions invoked are “function of both the event and the program state.”
- State “attributes” (variables) allow for sharing information between actions.
- Concurrent and nested state machines can flexibly represent complex applications without state explosion.
- Efficient use of RAM without explicit memory management.
Manual stack management: \texttt{sum} and \texttt{num} are only used during sampling

```c
int sum = 0;
int num = 0;
bool sampling_active=false;

void init_remote_compare() {
    sampling_active=true;
    request_remote_temp();
    register_timeout( 5 );
}

void message_hdl( MSG msg ) {
    if(sampling_active==false) return;
    sum = sum + msg.value;
    num++;
}

void timeout_hdl() {
    sampling_active=false;
    int val = read_temp();
    int average = sum / num;
    if( average > val ) /* ... */
}
```

Manual Flow Control: sampling\_active value dictates execution across the entire program
Basic Concepts

- FSM is a set of states, transitions, and events.

- Each transition is associated with an event and connects two states.

- When an event occurs, transition associated with it is followed to a new state (subject to satisfaction of some predicate).

- Extend this concept to include hierarchy and concurrency.

- Also, allow states (and state hierarchies) to have local variables.

![State Machine Diagram](image)

**Fig. 1.** A sample state machine.

**Fig. 2.** Discrete time.
Parallel Composition

- Multiple state machines exist side by side

- When an event $e$ occurs with associated transitions in more than one state machine, all of them act on it.

- Event queue is used to convert real-world continuous time into discrete time steps.
Hierarchical Composition

- States can contain substates.
- Variables accessible to a state are accessible to its substates.
- Not 100% clear on how transitions are resolved when an event which is handled by a substate leads to the termination of that machine.
So, back to the beef.

Manual Stack Management

```c
state C {
    int c;
}
state A { int a1, a2; e / outA(c) -> B; }
state B { int b1, b2; f / -> A; }
} || state D {
    int d;
}
```

which results in the following memory layout in C:

```c
struct parCD {
    union machineC {
        struct stateC {
            int c;
        union machineAB {
            struct stateA { int a1, a2; } _stateA;
            struct stateB { int b1, b2; } _stateB;
        } _machineAB;
    } _stateC;
    } _machineC;
    union machineD {
        struct stateD {
            int d;
        } _stateD;
    } _machineD;
} _parCD;
```

`sizeof(parCD)=16 bytes`

```c
sizeof(c) + sizeof(a1) + sizeof(a2) 
+ sizeof(b1) + sizeof(b2) + sizeof(d) = 24 bytes
```
The Beef Goes On

Manual Flow Control

```c
initial state FREE {
    heartbeat_ev / -> FOLLOWER;
    sense_ev / -> MEMBER;
}

state FOLLOWER {
    sense_ev / -> MEMBER;
    entity_timeout_ev / -> FREE;
    heartbeat_ev / reset_entity_to() -> self;
    FREE -> heartbeat_ev / set_entity_to();
}

state MEMBER {
    FREE -> sense_ev / set_heartbeat_to(),
    start_leader_election();
    FOLLOW -> sense_ev / set_heartbeat_to();
    heartbeat_ev / send_position(),
    send_heartbeat() -> self;
    lost_ev / -> FOLLOWER;
    elected_ev / -> LEADER;
}

state LEADER {
    position_type pos;
    lost_ev / start_leader_election() -> FOLLOWER;
    position_ev / aggregate_pos(pos, position_ev)
        -> self;
}
```
What’s missing?

* Interrupts violate this nice discrete event queue, but are
darn useful.

* As they point out, “meta-events” (i.e. “events e1, e2, e3 in
any order”) are really cumbersome.

* I think that it would be fabulous if common behaviors
could be abstracted (i.e. overriding functions in subclasses,
calling superclass implementations explicitly)

* What else is missing? What are some problems?
Why I liked this paper

- Good balance between flexibility and structure
- Attractive memory usage benefits
- Maps really nicely to the way that I think about design
- Natural transition from event-based model: don’t need to quit “cold turkey”
- Lightweight framework: not much more than an event queue.
- FSMs and verification go together like PEAS AND CARROTS
Can we implement this as an extension to nesC?

* Event queue is already in place to some extent (though it lacks an explicit concept of concurrency/discrete timeline (does that matter?))

* Component-based programming model and parameterized interfaces translate beautifully: state is stored as a single global variable, highly generic façade modules call correct state-specific event handlers.

* Interrupts/async code might be tricky to work in. They might remain largely outside of this model.