BEADL: A New Real-Time Language for Behavioral Experiments

Stephen A. Edwards    John Hui    Maryam Husain

Columbia University

December 10, 2018
Adam Kepecs, Cold Spring Harbor Laboratory

[Lak et al., Neuron 84(1), 2014]
Bpod: An Open Hardware Platform for Behavioral Monitoring and Control

Sanworks.io, spun out of Kepecs’ lab.
Teensy 3.6: ARM Cortex M4, 180 MHz
sma = NewStateMatrix();
sma = AddState(sma,'Name', 'ITI',
   'Timer', S.ITI,
   'StateChangeConditions', {'Tup', 'PreState'},
   'OutputActions',{});
%Pre task states
sma = AddState(sma,'Name','PreState',
   'Timer', S.GUI.PreCue,
   'StateChangeConditions', {'Tup', 'CueDelivery'},
   'OutputActions', {'BNCState',1});
%Cue
sma=AddState(sma,'Name', 'CueDelivery',
   'Timer', S.GUI.CueDuration,
   'StateChangeConditions', {'Tup', 'Delay'},
   'OutputActions', {'SoftCode',S.Cue});
%Delay
sma=AddState(sma,'Name', 'Delay',
   'Timer', S.Delay,
   'StateChangeConditions', {'Tup', 'ExtraCueDelivery'},
   'OutputActions',{});
%Extra Cue for L3-SecondaryCue
sma=AddState(sma,'Name', 'ExtraCueDelivery',
   'Timer', S.ExtraCueDuration,
   'StateChangeConditions', {'Tup', 'ExtraDelay'},
   'OutputActions',{});
%Extra Delay for L3-SecondaryCue
sma=AddState(sma,'Name', 'ExtraDelay',
   'Timer', S.ExtraDelay,
   'StateChangeConditions', {'Tup', 'Outcome'},
   'OutputActions',{});
%Reward
sma=AddState(sma,'Name', 'Outcome',
   'Timer', S.Outcome,
   'StateChangeConditions', {'Tup', 'PostOutcome'},
   'OutputActions', {'ValveState', S.Valve});
%Post task states
sma=AddState(sma,'Name', 'PostOutcome',
   'Timer', S.GUI.PostOutcome,
   'StateChangeConditions', {'Tup', 'exit'},
   'OutputActions',{});
SendStateMatrix(sma);

---

**Describe as FSM**

**Build FSM with Matlab API**

**Download FSM to firmware**

**Firmware: FSM interpreter w/100 µs heartbeat**
BEADL: The Idea

**Controller** → **Stimuli** → **Subject** → **Responses** → **BEADL** → **Results**

**outputs**
- `valve dispense` # Channel w/ 1 event
- `led on off` # Two possible events

**inputs**
- `gate enter exit`

**task simple_example:**
- "Subject Attraction" # State label
- `valve dispense` # Generate event
- `await 10 s:` # Timeout
  - "Failure"
  - `goto "Subject Attraction"` # Event arrived
- "Light Stimulus"
  - `led on` # Generate event
  - `await 100 ms` # Simple delay
  - `led off`
Deterministic formal semantics

Explicit model-time delays only; platform-independent timing above some minimum delay (synchronous logic)

“Bare metal” microcontroller implementations: hardware counter/timer drives timing, timer interrupts for scheduling

Schedulability/static timing analysis done at compile time
BEADL: Possible Single-Threaded Implementation

void interrupt1() {
    now = get_platform_time();
    switch (state) {
    case S1:
        Attract: report("Attract");
            valve_dispen();
            state = S2, schedule(now + SEC(10)); return;
    case S2:
        switch (get_interrupting_event()) {
            case TIMEOUT:
                Fail : report("Fail");
                    goto Attract;
            case GATE_ENTER: break;
            default: return; }
        Stimulus: report("Stimulus");
            led_on();
            state = S3, schedule(now + MS(100)); return;
    case S3:
            led_off();
            state = STOPPED; return;
    case STOPPED:
            return;
    }
}
BEADL: Parallel Composition

Language Design is Library Design —Stroustrup

A desired BEADL library: input debounce

Nervous rats often jitter before making a decision; want a library that discards “on” events shorter than $x$ ms

⇒ Parallel composition?

Feedback loops?
Simultaneous events?
Contradictions?
“Real-time discrete-event simulation”
Parallel actors with multiple inputs, outputs, channels

Event tags: $\langle$model time, microstep, topological level$\rangle$
Feedback loops must have delay actors (microstep or longer)
Scheduler w/ event queue; timer interrupt delays keep model time “close” to platform time.

[Edward Lee et al.]
node Watchdog (set, reset, u_tps: bool; delay: int) returns (alarm: bool);
var is_set : bool;
    remain : int;
let
    alarm = is_set and (remain = 0) and pre(remain) > 0;
    is_set = false -> if set then true
    else if reset then false
    else pre(is_set);
    remain = 0 -> if set then delay
    else if u_tps and pre(remain) > 0
    then pre(remain) − 1

tel

Declarative dataflow style; expressing control is awkward

Every loop must have a unit delay (“pre”)

Implicit clock not tied to wallclock time

[Halbwachs, Caspi, et al.]
The Esterel Synchronous Programming Language

module ABRO:
input A, B, R;
output O;

loop
    [ await A || await B ];
emit O
each R

end module

Imperative style with sequencing, concurrency, conditionals, and exceptions

More subtle causality constraints; “constructive” causality requires a per-state analysis

[Berry et al.]
BEADL: A Work in Progress

- **Semantics**
  Event-driven with explicit model time advances
  Synchronous: Esterel-like with Lustre-style causality?
  No reactions to absent events (timeouts only)

- **Implementation**
  PTIDES-like: Interrupt driven
  Events: timeouts, input arrivals
  Model time “matched” to platform time

- **Schedulability**
  Run-time deadline checking
  Compile-time WCET analysis?