From Simulation to Execution: on a Certain Programming Paradigm for Reactive Systems

Pawel Gburzynski

Department of Computing Science
University of Alberta
&
Olsonet Communications
SMURPH

A System for Modeling Unslotted Real-time PHenomena

\[ t_a \]

\[ 2t_a \]
Objects

link
port
station

processes

packets, traffic patterns, mailboxes, timers, ...
A sample code fragment

```c
mailbox DBuffer (Packet*) {
    void outItem (Packet *p) { delete p; };
    void setup () { setLimit (MAX_Long); };
};

...

station PMonitor : PMonitorInterface {
    DBuffer *DB;
    void setup () {
        PMonitorInterface::configure ();
        DB = create DBuffer;
    }
};
```
A process

```
process SlotGen (PMonitor) {
    Port *ORing;
    DBuffer *DB;
    SMarker *SMark;
    int SCount;
    void setup () {
        ORing = S->ORing;
        DB = S->DB;
        SMark = create SMarker;
        SMark->fill (NONE, NONE, SlotML);
        setFlag (SMark->Flags, TOKEN);
        SCount = 0;
    }
    states {GenSlot, XDone};
    perform;
}
```
A process

SlotGen::perform {

    state GenSlot:
        if (DB->nonempty ())
            terminate;
        else {
            SMark->Number = SCount;
            ORing->transmit (SMark, XDone);
        }

    state XDone:
        ORing->stop ();
        ++SCount;
        Timer->wait (SegmWindow, GenSlot);
};
SMURPH -> SIDE

Interface to outside world: mailboxes can be mapped to devices or sockets.

Virtual time mapped to real time. This simply means that Timer delays are “actual.”
A natural paradigm

Mailbox Sensor {
    ...
    int getValue ();
    ...
}

Mailbox Actuator {
    ...
    int setValue ();
    ...
}

Sensor *sns;
 ...
    sns->wait (RECEIVE, ...);
 ...

How do we program small devices?

From scratch: theoretically most efficient, practically tedious and/or messy (certainly not very effective).

Some platforms do exist, e.g., TinyOS. Unfortunately, they are quite a bit into consortium-type solutions.

What is the problem? Why can't we simply use Java (just kidding)? 😊
The inspiring project

To implement a simple wireless application on a card-sized microcontrolled device.

Two stages:

• High-level design and virtual implementation in SMURPH - to verify the concept and test performance.
• The port of SMURPH code to the micro (by hand).
Multiple processes/threads

How to have threads without a stack for each of them?

E.g., TinyOS has no threads, only:

- callbacks (event handlers) and
- “tasks” (non-preemptible chunks of code)
process (sniffer, sess_t)
    char c;
    entry (RC_TRY)
        data->packet = tcv_rnp (RC_TRY, efd);
        data->length = tcv_left (packet);
    entry (RC_PASS)
        if (data->user != US_READY) {
            wait (&data->user, RC_PASS);
            delay (1000, RC_LOCKED);
            release;
        }
    ...
    entry (RC_LOCKED)
    ...
    entry (RC_ENP)
        tcv_endp (data->packet);
        signal (&data->packet);
        proceed (RC_TRY);
endprocess (1)
PicOS's footprint

- 20 bytes of RAM per process
- 64 bytes of (global) stack goes a long way
- We have a non-trivial mesh system based on MSP430F148 (with 2KB of RAM per node)
TCV plugin model

application

TARP

MAC

API

pluggable transceiver interface (TCV)

plugs

phy

the kernel

the micro

radio

radio

TCP/IP?
Plugin interface

```c
typedef struct {
    int (*tcv_ope)(int, int, ...);
    int (*tcv_clo)(int, int);
    int (*tcv_rcv)(int, address, int, int*, tcvadp_t*);
    int (*tcv_frm)(address, int, tcvadp_t*);
    int (*tcv_out)(address);
    int (*tcv_xmt)(address);
    int (*tcv_tmt)(address);
    int tcv_info;
} tcvplug_t;
```

- **preprocessing upon reception**
  - how to open a session
  - how to close a session
  - preprocessing for output
  - application packet boundary
  - after packet transmission
  - on timeout