A Dataflow Approach to the ROS Architecture

Orestis Melkonian

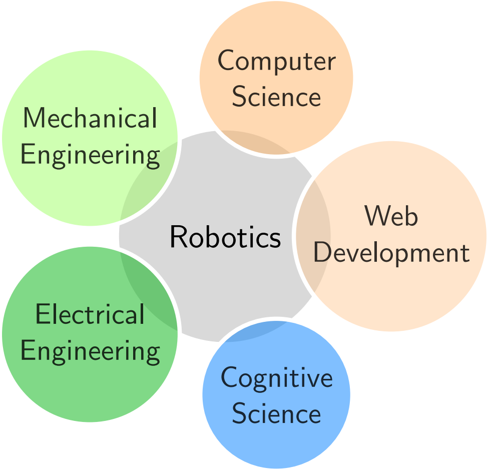
Software & Knowledge Engineering Laboratory (SKEL)

o.melkonian@di.uoa.gr

December 27, 2015

Overview

Introduction



Future

- Ubiquitous Robotic Systems
- Web Robotics
- Minimum-cost robotics research
- Rapid prototyping

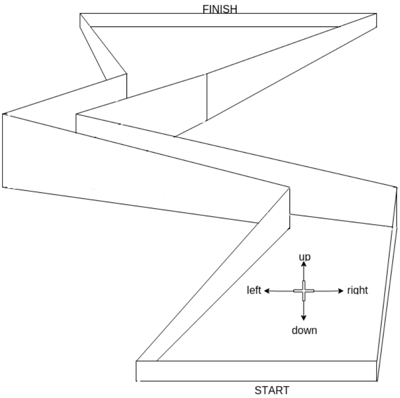
Motivation

- Satisfied with current languages on ROS?
- Right amount of abstraction?
- Logical perspective in sync with code we write?
- Real-time systems
  - Do existing languages provide time modelling?
- Continuous streams of data
  - Do existing languages provide appropriate stream handling?
- Too much technicalities that are not part of the main problem
  - Maybe a compiler can do them automatically
- Do problems faced in robotics have a dataflow nature?
  - If so, why not code in a dataflow language?

Example I

A drone progressing in a hall

- 4 sonar sensors (UP,DOWN,LEFT,RIGHT)
- Default move: AHEAD
- Depending on sensor data, change direction to avoid collision with any of the four walls
- Also occasional sonar sensor malfunction (negative values)

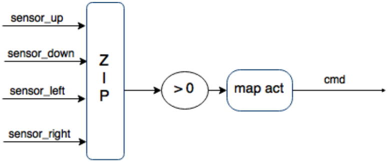


Example II

```
float arr[4] = {0, 0, 0, 0};
void callback_up(float val) { if (val > 0) arr[0] = val; }
void callback_down(float val) { if (val > 0) arr[1] = val; }
void callback_left(float val) { if (val > 0) arr[2] = val; }
void callback_right(float val) {
        if (val > 0) arr[3] = val;
        string cmd_ver = "", cmd_hor = "";

        if (arr[0] > arr[1]) cmd_ver = "UP"
        else if (arr[0] < arr[1]) cmd_ver = "DOWN"
        else cmd_vertical = ""
        if (arr[2] > arr[3]) cmd_hor = "LEFT"
        else if (arr[2] < arr[3]) cmd_hor = "RIGHT"
        else cmd_hor = ""

        ROS.publish("cmd", cmd_ver + cmd_hor);
}
int main() {
        ROS.subscribe("sensor_up", callback_up); ROS.subscribe("sensor_down", callback_down);
        ROS.subscribe("sensor_left", callback_left); ROS.subscribe("sensor_right", callback_right);
        ROS.publisher("cmd");
        ...
}
```

Example III

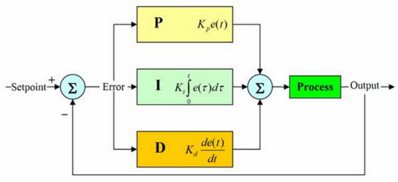
```
Topic fused = zip
                (subscribe "sensor_up",
                subscribe "sensor_down",
                subscribe "sensor_left",
                subscribe "sensor_right")
                >>> filter (> 0)

Node actor = publish "cmd" (map act fused)

act (u,d,l,r) = ver ++ hor where
        ver = if u>d "UP" else (if u<d "DOWN" else "")
        hor = if l>r "LEFT" else (if l<r "RIGHT" else "")
```



Another Example

Implementation of a PID controller

```
constant Kp, Ki, Kd, setpoint
Graph g = loop(output):
        error = setpoint - output
        p = (*Kp) <<< error
        i = (*Ki) <<< integral <<< error
        d = (*Kd) <<< derivative <<< error
        result = sum [p, i, d]
        output <<< process <<< result
```



Problems I

- **Scalability**
  - Asynchronous behaviour using callback functions
  - Complex schemes require "internal plumbing"
  - Unreadable, difficult to maintain
  - Does not separate data from control
- **Untyped topics**
  - Topics are just named pipes
  - No type-safety
  - Hard to impose constraints on them
  - e.g. Visual detection on audio streams
    - Does not catch errors at compile time

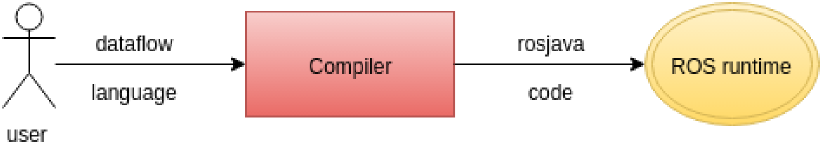
Problems II

- **Time modelling**
  - Control theory models robot behaviour primarily using differential equations on time
  - Existing languages do not provide the concept of time
  - Need to implement tedious delta timing
  - But why? Time is logical
  - Need to have temporal nature by design
- **Compiler restriction**
  - Coder explicitly specifies where nodes are being executed
  - Won't allow intelligent runtime re-configurations
    e.g. for optimal power management, minimum execution time

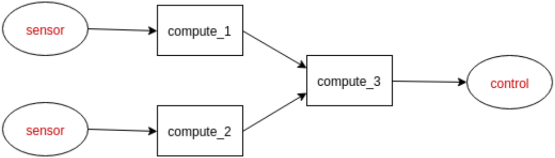
Problems III

- **Verifiability**
  - Robots are becoming an integral part of our everyday life
  - More and more responsibility
    e.g. babysitters, security monitoring, space robots
  - Must be able to prove program correctness
  - Nearly impossible for low-level languages
  - Need for more precise semantics

Solutions I

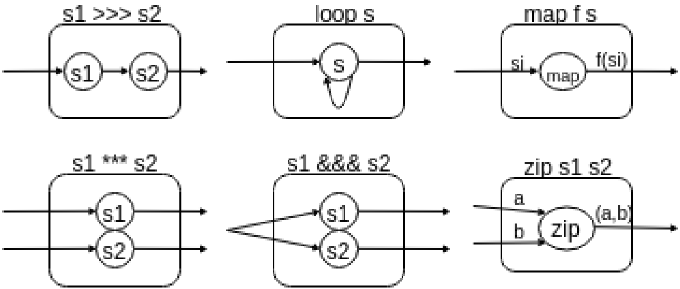


- **A domain-specific dataflow language**
  - ROS architecture essentially defines a dataflow graph



Solutions II

- Composable stream-processing operators



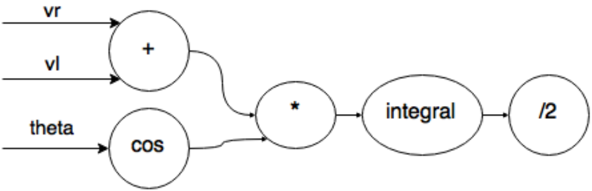
Solutions III

- Timely model $\rightarrow$ Simpler, portable code
  e.g.

$$x(t) = 1/2 \int_0^t (u_l(t) + u_r(t)) cos\theta(t) dt$$

$$\Downarrow$$

```
xs = (v &&& t) >>> * >>> integral >>> /2
where    v  = (vr &&& vl) >>> +
         t  = theta >>> cos
```



Solutions IV

- Separates data from control

```
void handle_data(float val) {
        if (val > threshold)
                publish("cmd", act(val));
}
int main() {
        ROS.subscribe("sensor", handle_data);
        ROS.publish("cmd");
        ...
}
```

$$\Downarrow$$

```
Topic myTopic = subscribe "sensor" >>> filter(> threshold)
Node actor = publish "cmd" (map act myTopic)
```

Solutions V

- **Topics as streams**
  - First-class citizens
  - At last, type-safety!

```
Topic t :: Stream[Float]
Topic t = subscribe "sonar1" >>> filter(> 0)
```

  - Automatic *rosmsg* generation
  - We now can catch errors at compile time
- **Pure functions to the rescue**
  - Keep maximum portion of the program as pure functions
  - No side-effects → equational reasoning
  - Can prove program correctness

Solutions VI

- **Dynamic reconfiguration**
  - Locality-agnostic node execution
  - System will allocate nodes to machines for optimality
  - Ability to specify preferences
    e.g. run 'graph' –powersave/–uniform /–mintime
- **Harness the cloud!**
  - Adopt the advantages of web-based solutions to develop robotic applications
  - Use the cloud to get access to distributed computing resources
    e.g. Grasping using the Google Object Recognition Engine

Functional Reactive Programming (FRP)

- A programming paradigm suitable for hybrid systems
- Uses the building blocks of functional programming (*map*,*filter*,...)
- For systems that are:
  - Responsive
  - Resilient
  - Elastic
  - Message-driven
- Explicit time modelling
- Express event-handling in a natural way
- Applications in GUIs, robotics and music.

Robotics

- **roshask**
  - A Haskell-binding client library for ROS
  - Takes the first step towards stream-oriented robotics
  - Implements topics as first-class citizens
  - Provides some basic stream manipulation
- **Frob**
  - A domain-specific language (DSL)
  - Realization of the FRP model on robotics
  - Complex robot behaviours in a few lines of code
  - Implementation exists in Haskell (Yampa)
  - Several performance issues (time-/space-leaks)

Big Data

- **Akka**
  - A framework for scalable, distributed, message-driven applications
  - Incorporates the Actor model, like ROS
  - The stream handling seemed to be error-prone and tedious
  - The inability to treat streams efficiently and intuitively led them to develop the Akka Streams API
  - Also influenced by FRP
- **Naiad**
  - A distributed system for cyclic parallel dataflow programming
  - A new computational model: *Timely dataflow*
  - Many novel approaches for efficient dataflow graphs

Ziria: A DSL for wireless systems programming

- Designed by Haskell-lovers
- Also inspired by Functional Reactive Programming (FRP)
- Towards Software-Defined Radios (SDR)
- In contrast to SORA (low-level C++ library)
- High-level functional language replaces low-level C
- Implementation of a Wifi Receiver
    - SORA $\rightarrow$ 23000 lines of code
    - Ziria $\rightarrow$ 3000 lines of code
- Implementation of a Scrambler
    - SORA $\rightarrow$ 90 lines of code
    - Ziria $\rightarrow$ 20 lines of code
- But same performance!

Conclusion

- There will be some learning curve
- But it will pay off in terms of:
  - Productivity
  - Cleaner, safer, scalable code
  - More efficient resource management
  - Faster robotics research
- Any DSL design relies heavily on feedback from domain experts
- There is a general pattern here
  - Abstraction is the way!

The End