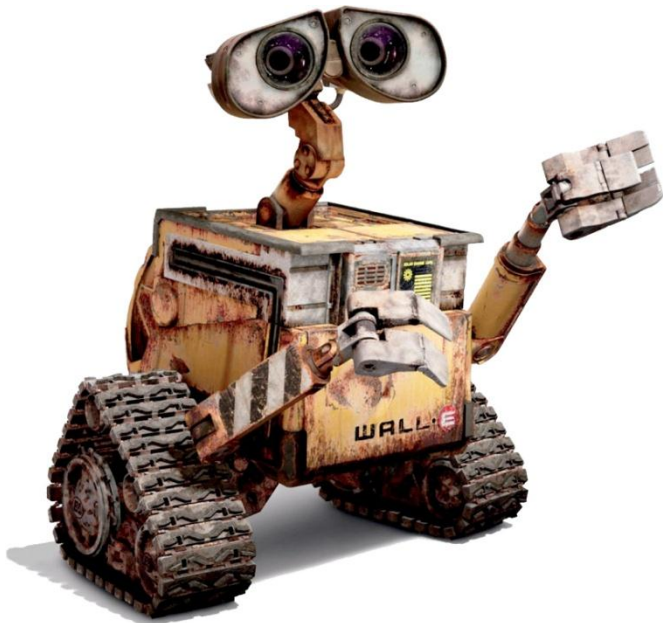


REACT



REACT:

Event-driven

Asynchronous

Concurrent

Turing-complete

Students: Will Noble,
Aleksandar Milicevic

Supervisor: Stelios Sidiroglou

PI: Prof. Martin Rinard

Robotics Programming

- Abstraction gap
 - ▣ high-level problem, low-level implementation primitives
- Typically steep learning curve, especially for hobbyists
- Programming in terms of low-level primitives is tedious and error-prone
- Largely non-standardized



Distributed, Interactive, Heterogeneous

- Concurrent and distributed architecture

- data races
- atomicity
- deadlocks
- shared data inconsistency

- Implementation complexity

- hard to analyze, test, ensure correctness



Proposed Solution

- **Model-based, event-driven paradigm**
 - global model of the entire distributed system
 - simple sequential semantics
 - expressive programming language
- **Runtime environment**
 - manages access to shared state
 - no data races by construction
- **Analyses**
 - amenable to formal analyses (e.g., testing, security, ...)

REACT: Records, Contexts, Events

□ Records

- simple data structures
- used to represent the core data model of the system

□ Contexts

- encapsulate different processes (nodes)

□ Events

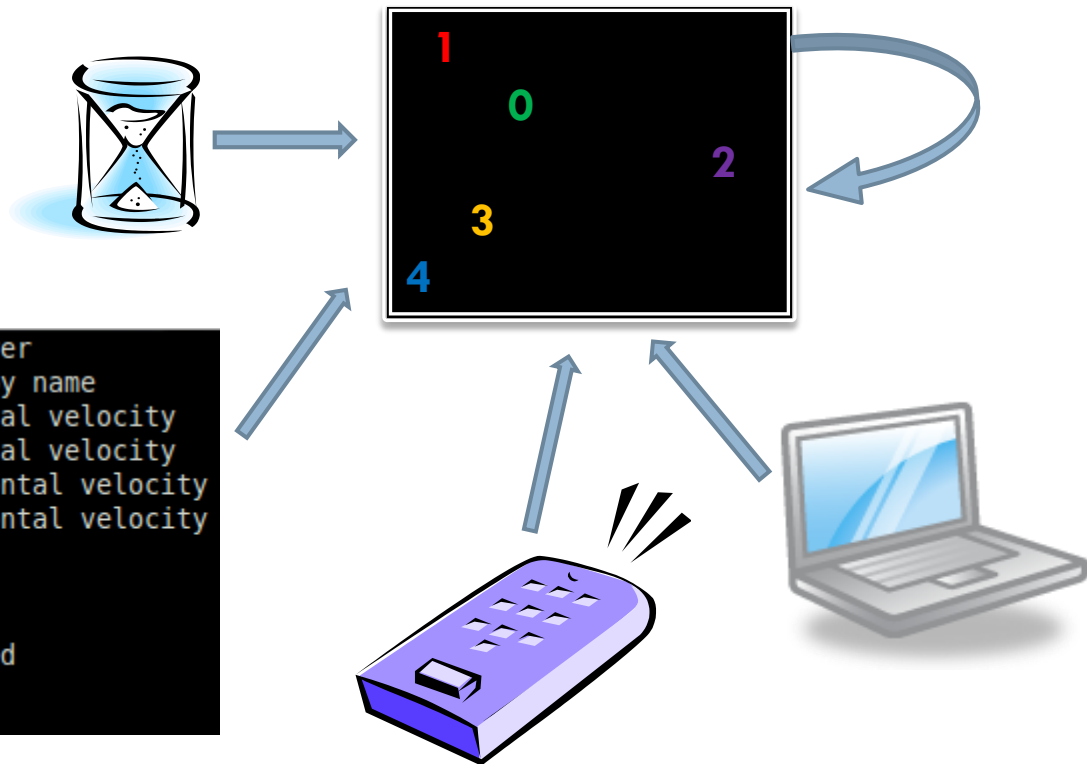
- allow robots to dynamically react to their environments
- triggered by the user, timer, whenever a condition holds, ...

Example: BeaverSim

- **Implement a beaver simulator:**
(inspired by the ROS turtlesim example)
 - *model*: a beaver has position (x, y) and speed (v_x, v_y)
 - *constraint*: no more than 5 beavers allowed
 - *every* 1s positions are updated according to speed
 - *whenever* a beaver hits a wall, its speed is reversed
 - one simulator node displays current positions of all beavers
 - arbitrary number of remote controller nodes

Implementation challenges

- concurrent access to (shared) beaver data
 - ▣ from multiple remote controllers
 - ▣ from timer events
 - ▣ from GUI threads



```
c          - create new beaver
0..5      - select beaver by name
key_up    - decrease vertical velocity
key_down  - increase vertical velocity
key_left  - decrease horizontal velocity
key_right - increase horizontal velocity

selected beaver: 3

guard for event Spawn failed
Too many beavers
```

Traditional approach to timer events

- fragmented implementation of *whenever* actions
 - *whenever* conditions can turn true at various code points
 - e.g., (1) *when position is auto-updated based on speed and*
(2) *when position is explicitly set by a remote controller*
- fragmented implementation of *constraint* checks
 - have to make sure that invariants hold after every update

REACT: domain-specific features

conditional events

```
whenever (condition) {  
  [code to execute  
  whenever the  
  condition is true]  
}
```

typed events

```
on EventType {  
  [code to execute when  
  an event of the  
  above type occurs]  
}
```

periodic events

```
every (interval) {  
  [code to execute  
  every interval ms]  
}
```

invariants

```
invariant {  
  [condition that must  
  hold at all times]  
}
```

BeaverSim in REACT: model

```
MAX_BEAVERS = 5
MAX_X, MAX_Y = (10, 10)
```

```
record Beaver [  
    name: str,  
    x: int,  
    y: int,  
    vx: int,  
    vy: int  
]  
  
context BeaverSim [  
    beavers: listof(Beaver)  
] {  
    invariant {  
        beavers.size() < MAX_BEAVERS  
    }  
    # ...  
}  
  
context RemoteCtrl {  
    # ...  
}
```

BeaverSim in REACT: events

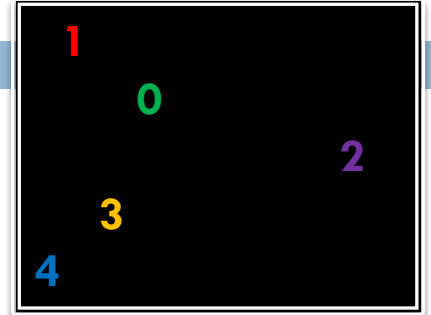
```
event Spawn [
  receiver: BeaverSim,
  name:     str
] {
  guard    { name.length() == 1 }

  handler {
    receiver.beavers += Beaver.new(name: name,
                                   x: 0, y: 0,
                                   vx: 1, vy: 0)
  }
}
```

BeaverSim in REACT: events

```
event ChangeSpeed [  
  receiver: BeaverSim,  
  idx:      int,  
  dx:      int,  
  dy:      int  
] {  
  guard    { 0 <= idx < receiver.beavers.size() }  
  
  handler {  
    receiver.beavers[idx].vx += dx  
    receiver.beavers[idx].vy += dy  
  }  
}
```

BeaverSim in REACT: contexts



```
context BeaverSim [  
  beavers: listof(Beaver)  
] {  
  on_start      { @gui = MyGui.new; @gui.start() }  
  on_exit       { @gui.stop() }  
  
  every(1000) {  
    @gui.draw_beavers(beavers)  
    for b in beavers { b.x += b.vx; b.y += b.vy }  
  }  
  
  whenever(some b in beavers | b.x < 0) {  
    b.x = 0; b.vx = -b.vx  
  }  
}
```

BeaverSim in REACT: contexts

```
context RemoteCtrl {
  on start { @selected = -1 }

  on KEY_0 { @selected = 0 }
  on KEY_4 { @selected = 4 }

  on KEY_c      { trigger Spawn.new(name: 'B') }

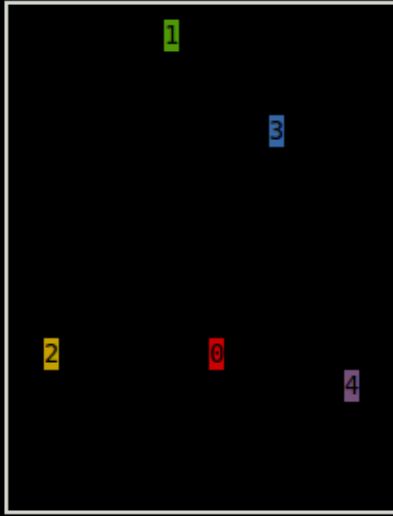
  on KEY_UP     { trigger ChangeSpeed.new(idx: @selected
                                           dx: 0, dy: -1) }
  on KEY_DOWN   { trigger ChangeSpeed.new(idx: @selected
                                           dx: 0, dy: 1) }
  on KEY_LEFT   { trigger ChangeSpeed.new(idx: @selected
                                           dx: -1, dy: 0) }
  on KEY_RIGHT  { trigger ChangeSpeed.new(idx: @selected
                                           dx: 1, dy: 0) }
}
```

```
c          - create new beaver
0..5      - select beaver by name
key_up    - decrease vertical velocity
key_down  - increase vertical velocity
key_left  - decrease horizontal velocity
key_right - increase horizontal velocity

selected beaver: 3

guard for event Spawn failed
Too many beavers
```

Demo (implemented on top of ROS)



```
c          - create new beaver
0..5      - select beaver by name
key_up    - decrease vertical velocity
key_down  - increase vertical velocity
key_left  - decrease horizontal velocity
key_right - increase horizontal velocity
```

selected beaver: 4

successfully executed ChangeSpeed event

```
c          - create new beaver
0..5      - select beaver by name
key_up    - decrease vertical velocity
key_down  - increase vertical velocity
key_left  - decrease horizontal velocity
key_right - increase horizontal velocity
```

selected beaver: 0

successfully executed ChangeSpeed event

Original TurtleSim Spawn event

```
spawn_srv_ = nh_.advertiseService("spawn", &TurtleFrame::spawnCallback, this);
bool TurtleFrame::spawnCallback(turtlesim::Spawn::Request& req, turtlesim::Spawn::Response& res) {
    std::string name = spawnTurtle(req.name, req.x, req.y, req.theta);
    if (name.empty()) { ROS_ERROR("A turtle named [%s] already exists", req.name.c_str()); return false; }
    res.name = name;
    return true;
}

std::string TurtleFrame::spawnTurtle(const std::string& name, float x, float y, float angle) {
    std::string real_name = name;
    if (real_name.empty()) {
        do {
            std::stringstream ss;
            ss << "turtle" << ++id_counter_;
            real_name = ss.str();
        } while (hasTurtle(real_name));
    } else { if (hasTurtle(real_name)) { return ""; } }
    TurtlePtr t(new Turtle(ros::NodeHandle(real_name), turtle_images_[rand() % turtle_images_.size()], QPointF(x, y),
angle));
    turtles_[real_name] = t;
    update();
    ROS_INFO("Spawning turtle [%s] at x=[%f], y=[%f], theta=[%f]", real_name.c_str(), x, y, angle);
    return real_name;
}
```


Original TurtleSim model class

```
class Turtle {
public:
    Turtle(const ros::NodeHandle& nh, const QImage& turtle_image, const QPointF& pos, float orient);
private:
    void velocityCallback(const geometry_msgs::Twist::ConstPtr& vel);
    bool teleportRelativeCallback(turtlesim::TeleportRelative::Request&, turtlesim::TeleportRelative::Response&);
    bool teleportAbsoluteCallback(turtlesim::TeleportAbsolute::Request&, turtlesim::TeleportAbsolute::Response&);
    ros::Subscriber velocity_sub_;
    ros::Publisher pose_pub_;
    ros::ServiceServer teleport_relative_srv_;
    ros::ServiceServer teleport_absolute_srv_; }

namespace turtlesim {
Turtle::Turtle(const ros::NodeHandle& nh, const QImage& turtle_image, const QPointF& pos, float orient)
: nh_(nh), turtle_image_(turtle_image), pos_(pos), orient_(orient), lin_vel_(0.0), ang_vel_(0.0), pen_on_(true),
pen_(QColor(DEFAULT_PEN_R, DEFAULT_PEN_G, DEFAULT_PEN_B)) {
    velocity_sub_ = nh_.subscribe("cmd_vel", 1, &Turtle::velocityCallback, this);
    pose_pub_ = nh_.advertise<Pose>("pose", 1);
    teleport_relative_srv_ = nh_.advertiseService("teleport_relative", &Turtle::teleportRelativeCallback, this);
    teleport_absolute_srv_ = nh_.advertiseService("teleport_absolute", &Turtle::teleportAbsoluteCallback, this);
}
```

Big Idea

- Generic platform for programming event-driven systems
 - covers a whole class of programs

interactive event-driven apps



REACT

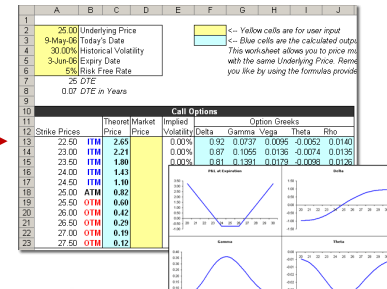
similar to



cms



spreadsheets



- End-user programming of interactive apps
 - examples: social web apps, robots
 - makes simple tasks easy and difficult ones possible

Status

- Prototype for **client/server** applications
 - ▣ implemented in Java



- Prototype for **web applications**
 - ▣ implemented for Ruby on Rails



- Next: look for concrete robot examples
 - ▣ robots are event driven, often mission critical
 - ▣ adapt our paradigm to programming robots
 - ▣ verify functional correctness



Benefits and Future Goals

- Rich programming platform

- speeds up development process

- eliminates a whole class of concurrency bugs by construction

- Application in the security domain

- every field access is managed by the runtime system

- security policies can be defined independently and automatically enforced at runtime

- Robot programming for end-users



Thank You!

The End



Hello World example

```
context Main {  
  /* trigger-event */  
  on Main:enter {  
    /* action call w/ argument */  
    Sys.print! msg: "Hello, world!"  
    /* built-in action call */  
    Main.exit!  
  }  
}
```

Outputs: Hello, world!

A more complex example

```
context Headbanger {
  banging = 0
  bangSpeed = 0

  action bangHead! forTime:dur:5000 withEnthusiasm:enth {
    banging = Clock.time + dur
    bangSpeed = enth
  }
  whenever (banging > Clock.time) {
    #spinhead(bangSpeed)
  }
}
context Main {
  on Main:enter {
    Headbanger.enter!
    Headbanger.bangHead! withEnthusiasm:10 forTime:10000
  }
  every (20000) {
    Headbanger.bangHead! withEnthusiasm:20
  }
}
```

Variables

Syntax:

```
(public) (active) name = value
```

where `name` is the variable identifier, `value` is a numerical expression

- `public` modifier allows variable to be visible outside of its own context
- `active` modifier creates an active variable: read-only once defined, and re-evaluate their assigned expression every time they are referenced. They are implemented as in-line function calls

In-depth: 'whenever' vs. 'every' events

Whenever

Syntax:

```
whenever (condition) {  
    [code to execute]  
}
```

- **condition: boolean expression to check**
- **for direct reactions to changes in the robot's environment**

Every

Syntax:

```
every(interval) {  
    [code to execute]  
}
```

- **interval: numerical expression for time interval**
- **requires some method of retrieving clock ticks**

Implementation:

```
last_call = 0  
whenever (last_call + interval  
    < clock_time) {  
    last_call = clock_time  
    [code to execute]  
}
```

In-depth: 'on' events vs. actions

'on' event

Syntax:

```
on cntxt_name:event_name {  
    [code to execute]  
}
```

Called explicitly with:

```
trigger cntxt_name:event_name
```

- for reactions to user-defined circumstances
- only execute if context is live

Action

Syntax:

```
action name! <arguments> {  
    code  
}
```

Argument syntax:

```
ext_name:int_name(:def_val)
```

- Use system of constraints to ensure safety
- Take dynamic arguments

Calling syntax:

```
context_id.action_id!  
    <arguments to pass>
```

Embedded C

- ❑ Special “C context” construct for creating libraries of C-code interfaceable with REACT, use `_c_context` keyword
- ❑ C contexts can contain active variables or actions.

```
_c_context Foo {  
    public active c_val = "<C expression>"  
    action c_act! withArg:arg:50 "  
        [code...]  
    "  
}
```

- ❑ Code copied verbatim from within quotes

In order to implement APIs for particular robots in REACT, platform-specific code will surely be needed. Embedding native C-code into REACT source can facilitate this.

Technical contributions



- *Expressive power & programming efficiency*
- *Programming language close to the problem domain*

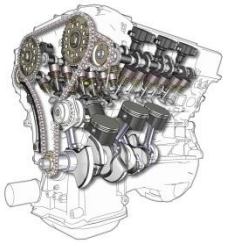
- think in terms of simple data structures
- don't worry about concurrency and distributed architecture
- declarative programming: say *what* not *how*

- **Runtime environment + code generation**

- no explicit synchronization, queues, message passing
- no data consistency issues
- synthesized clients for different platforms

- **Amenable to *tools, testing, and formal analyses***

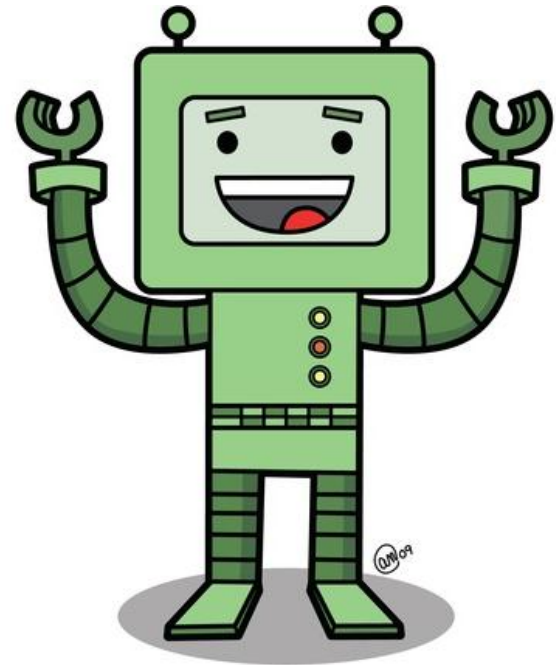
- core aspect of the system are kept succinct and formal
- important for safety/security critical systems



$$\frac{\sqrt{2}}{2} = \sqrt{\quad}$$

REACT

- Designed to be intuitive and easy to learn
- Powerful expressiveness
- Widespread applications



Proposed Solution

- Model-based, event-driven programming paradigm
 - ▣ provides a simple declarative conceptual model
 - ▣ expressive power & programming efficiency
 - ▣ programming language close to the problem domain
- Runtime environment
 - ▣ manages access to single shared global state
 - ▣ keeps everyone updated
 - ▣ programs free of concurrency bugs by construction
- Rich tool set
 - ▣ amenable to formal analyses and automated testing
 - ▣ enabled by the succinct and formal event model

REACT summary

□ Pros

- ▣ Highly abstract → easy to learn & portable
- ▣ Flexible → can interface with native C code
- ▣ Accessible → robotics programming requires extensive technical knowledge; REACT abstractions eliminate the need for hobbyists to acquire such knowledge.
- ▣ Expressive → programs written faster, robots developed more easily

□ Cons

- ▣ Centralized (not designed for distributed systems)
- ▣ Sequential implementation (no concurrent events)
- ▣ No explicit data model
 - ▣ data conflated with contexts

