International Journal of Robotic Computing
Vol. X, No. X (20XX) XXX-XXX

DOI: XXXXX

ROSY: An elegant language to teach the pure reactive nature of robot programming

Hugo Pacheco[1,2] and Nuno Macedo[1,3]

[1] INESC TEC, Porto, Portugal
{hugo.p.pacheco,nuno.m.macedo}@inesctec.pt
[2] FCUP, Porto, Portugal
[3] FEUP, Porto, Portugal

Received (xx/xx/xx)
Revised (xx/xx/xx)
Accepted (xx/xx/xx)

Abstract. Robotics is very appealing and is long recognized as a great way to teach programming, while drawing inspiring connections to other branches of engineering and science such as maths, physics or electronics. Although this symbiotic relationship between robotics and programming is perceived as largely beneficial, educational approaches often feel the need to hide the underlying complexity of the robotic system, but as a result fail to transmit the reactive essence of robot programming to the roboticists and programmers of the future.
This paper presents ROSY, a novel language for teaching novice programmers through robotics. Its functional style is both familiar with a high-school algebra background and a materialization of the inherent reactive nature of robotic programming. Working at a higher-level of abstraction also teaches valuable design principles of decomposition of robotics software into collections of interacting controllers. Despite its simplicity, ROSY is completely valid Haskell code compatible with the ROS ecosystem.
We make a convincing case for our language by demonstrating how non-trivial applications can be expressed with ease and clarity, exposing its sound functional programming foundations, and developing a web-enabled robot programming environment.

Keywords: robot programming; introductory programming; functional reactive programming.

1 Introduction

Robotics, though a multi-disciplinary area of engineering and science, shares a unique symbiotic relationship with computer science. As robots are increasingly

put to solve more complex tasks, the more important it becomes to look carefully into the software that runs inside them, and in particular to the programming languages that bring them to life. At the same time, programming a robot can grant computer science an often missing practical appeal, with a pedagogical potential long recognized by computer science educators [4].

Various studies corroborate that robotics can indeed mitigate the abstract nature of programming [18] and provide a more creative environment for teaching hands-on problem solving [10] and for coding rich behavior using basic code structures [13], while still requiring reasoning about relevant computational concepts such as software modularity and communication [15].

Yet, the richness of robot programming is synonym of its real-world complexity, which renders standard robotic frameworks unfeasible for use in a pedagogical setting and has been motivating the proposal of various educational robot programming frameworks. As the synergy between robotics and programming deepens, education is the perfect place to experiment novel approaches that can shape the robot programming languages of the future [7]. It also carries a timely opportunity to transmit good design practices to new generations of programmers, that are receptive to novel languages as long as these allow to quickly build applications [11].

As a step towards realising this vision, we advocate that languages for teaching robotics to novice programmers shall:

- be compatible with standard robotic practices and seamlessly connect to existing robotic infrastructures;
- adopt a simple declarative programming style and provide a pure cause-and-effect interface emphasizing the essence of what it means to program a reactive system;
- rely on a general-purpose programming language with good tool support, so that advanced programming features can be gradually introduced and acquired programming skills can naturally transfer to other domains.

In Section 2, we argue that state-of-the-art languages fail, to some extent, to exhibit these characteristics. This justifies the proposal of ROSY, a simple yet powerful reactive programming language, presented in Section 3. To ease adoption by novice programmers, ROSY is supported by a browser-based development environment, described in Section 4. As an embedded domain-specific language, ROSY supports the full power of higher-order functional programming offered by the host Haskell language, and is connected to ROS, one of the most popular robotic middlewares. The mechanics behind its implementation is presented in Section 5. Section 6 concludes the paper and leaves directions for future work.

This paper is an extended version of a conference version [19]. We have extended the ROSY language with the notion of parameters, services and actions, thus supporting most of the concepts of the ROS Computation Graph[4]. We introduce additional examples to demonstrate these new features and how they are integrated for the modular design of more complex, task-oriented, controllers.

[4] `http://wiki.ros.org/ROS/Concepts`

The environment has also been enhanced with a new simulation library, the TurtleSim, that demonstrates ROS services and the deployment of multiple identical robots, and allows for a gentler learning curve.

2 The pedagogy of robot programming languages

Programming robots is a particularly complex task, where one has to deal not only with the continuous and real-time aspects of the physical world, but also with heterogeneous architectures and complex communication paradigms. To minimize frustration of novice programmers, several pedagogical languages and approaches have been proposed over the years. This section reviews and discusses such related work.

2.1 The Robot Operating System

Robotic middlewares have been developed to ease the programming of robots, abstracting hardware and communication details and promoting modularity. The Robot Operating System (ROS) [23] is one such middleware, possibly the most popular, and defines an architecture through which components, called *nodes* in ROS, can communicate with each other. This communication can follow a many-to-many publish / subscribe paradigm through data sources called *topics*, or a request / reply paradigm through *services*. For longer and more complex tasks, ROS also provides an *action* library for non-blocking requests with periodic feedback. For all these functionalities, ROS allows the definition of custom message types, from which source code is automatically generated. A globally accessible key/value dictionary is also provided as a *parameter server*. Other than this, individual nodes are programmed in general-purpose languages, typically C++ or Python. The popularity of ROS is fueled by a very dynamic community and an open-source policy that encourages code re-use, and a large package database ranging from educational to industrial applications.

As a pedagogical example, consider the well-known TurtleBot2 robot[5], whose controller is programmed in ROS-powered C++. A main node controls its Kobuki mobile base publishing odometry information and data collected from a set of sensors (collision, cliff and wheel drop sensors, plus a few buttons), and subscribing to commands that control its (linear and angular) velocity and the color of a set of LEDs. Kobuki defines custom message types for the relevant events, from which C++ types are generated. Below is a minimal example of a ROS application that subscribes to bumper events and plays an error sound when a bumper is pressed:

Snippet 1 (Play a sound on collision).

```
ros::Publisher pub;

void cb (const kobuki_msgs::BumperEventConstPtr& b){
  if (b->state==kobuki_msgs::BumperEvent::PRESSED){
    kobuki_msgs::Sound s;
```

[5] https://www.turtlebot.com/turtlebot2/

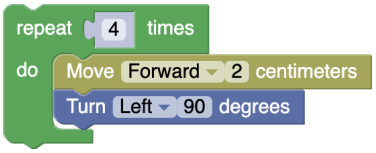


Fig. 1: A simple block to draw a square (TurtleBot3Blockly).

```
    s.value = kobuki_msgs::Sound::ERROR;
    pub.publish(s); } }

int main(int argc, char** argv){
  ros::init(argc, argv, "play");
  ros::NodeHandle nh;
  ros::Subscriber sub = nh.subscribe(
           "/mobile_base/events/bumper", 10, cb);
  pub = nh.advertise<kobuki_msgs::Sound>(
           "/mobile_base/commands/sound", 10);
  ros::spin();
  return 0; }
```

Even this minimal snippet is clearly non-trivial for novice programmers, obfuscating the truly reactive nature of the controller. Besides having to get around advanced linguistic features such as pointers, namespaces or templates, to manipulate topics the programmer must first explicitly `subscribe` to bumper events and `advertise` that sound commands will be published, considering the buffer size for incoming and outgoing messages. The programmer must also reason about how topics are processed, by registering a callback on the `ros::Subscriber` that will `publish` an error sound command if a bumper pressed event is read, and when topics are processed, in this simplest case using the ROS `spin` primitive that periodically processes callbacks for the queued messages.

ROS is widely used in educational contexts, and the community has developed considerable material to assist novice robotic programmers. One such popular library is the TurtleSim[6], which allows a client to control multiple minimalistic simulated robots, named "turtles", each subscribing to velocity commands. To manage the number of turtles being simulated, a set of ROS services is provided to spawn and kill turtles. ROS tutorials on actions usually also rely on the TurtleSim to exemplify longer running tasks, such as drawing geometrical shapes in multiple steps.

2.2 Visual robot programming languages

To tame the complexity of programming robots in fully-fledged general-purpose programming languages, a myriad of frameworks aimed at novice robot programming using visual programming languages have been proposed [9]. Many of these follow a block-based approach [2, 6, 16, 17, 26], where predefined blocks with varied colors and edges can be put together like pieces of a puzzle to define a robotic controller.

[6] `http://wiki.ros.org/turtlesim`

Aside from discussions on whether block-based approaches constitute "real" programming or their programming experience transfers to "real" textual languages [25,27], and with due exceptions such as [16], the vast majority of block-based robot programming approaches adopt an imperative mindset: primitive blocks execute individual predefined tasks, such as moving forward during a certain period of time or for a certain distance; and combining blocks amounts to performing sequences of tasks. For example, the TurtleBot3 can be programmed via a block-based interface illustrated in Fig. 1. Albeit simple, this example block hides away the reactive nature of the robotic system – all the sensor and command behavior is encapsulated within the black-box primitive blocks. Other non-pedagogical approaches such as SMACH[7] facilitate the design of complex robot controllers as hierarchical state machines, but each state/block is coded in standard ROS.

2.3 Functional reactive programming languages

Reactive programming [3] is a programming paradigm organized around information producers and consumers, that can naturally bring out the intrinsically reactive nature of cyber-physical systems. A particularly active line of research, known as functional reactive programming (FRP) [21], focuses on streams of information as the central reactive abstraction, and advocates a declarative approach to manipulate streams at a high-level of abstraction supported by the pure equational reasoning of functional languages such as Haskell.

Functional languages, as a form of algebra, are a good fit for introductory programming [11], and many pedagogical FRP approaches have been proposed for programming interactive games and animations [1, 5, 7, 12]. Much classical work has also proposed FRP for programming robots [14, 20, 22].

In FRP, time is typically explicit and conceptually continuous; the system is executed by sampling all streams synchronously at the rate of an external global clock. The ROSHASK [8] Haskell ROS library promotes the modularity and expressiveness of FRP, while remaining faithful to the asynchronous nature of ROS, by adopting a more pragmatic approach centered on manipulating topics in their entirety.

2.4 Towards a pedagogical robot FRP language

Despite the declarative nature of functional programming and the focus on events of reactive programming, the combinatorial FRP style is not beginner friendly, as it tends to swallow entire programs and resorts to advanced higher-order features to separate reactive code (referring to entire topics) from non-reactive code (referring to individual events). The success of existing pedagogical FRP approaches then lies on giving away some expressiveness and making time implicit, what liberates novice programmers from specific FRP syntax and invites them to simply write non-reactive pure functions, that are synchronously executed at a fixed rate.

[7] `http://wiki.ros.org/smach`

In this paper, we advocate that a toned-down FRP language, focused on individual events, captures the reactive essence and represents a sweet spot of introductory robot programming.

To support asynchronous behavior and retain some of the expressiveness of FRP, these functions then have an intuitive interpretation as operations on data streams.

3 The ROSY language

The ROSY language presents itself as a natural dialect for bringing robots alive using nothing more than plain mathematics, while promoting good software design practices. In this section, we make a case for how its declarative nature allows creating robotic controllers in an intuitive and painless way, and informally present its defining features through a collection of examples of increasing complexity. The ROSY language is independent of the robot drivers, but is statically wired to the communication topics of a particular robot. The programming model of ROSY focuses is built around the notion of ROS topics, and provides abstractions to model ROS-like services, actions or parameters not consider more advanced non-reactive ROS features such as services.

The ROSY website[8] offers a modern integrated development environment, including an editor, extensive documentation, help guides, and executable versions of all the examples shown in this section and more.

For readers not familiar with Haskell syntax, all the functions and operators not defined in this paper are standard and their definition can be found at `https://hoogle.haskell.org`. The documentation for respectively colored ROSY-specific functions and types is available at the ROSY website.

3.1 A ROSY primer

In ROSY, we can control a robot by writing pure functions that receive sensor information from the robot and react by sending commands back to the robot. To make a robot move forward at a constant velocity of 0.5 m/s, we can simply write:

Example 1 (Move forward).

```
move :: Velocity
move = Velocity 0.5 0

main = simulate (Kobuki Nothing) move
```

The `move` function models our controller (where the robot `Velocity` is separated into its linear and angular components), and the `main` function is ROSY-specific syntax to `simulate` our controller. (that will be elided from now on). The ROSY language supports a Kobuki or a TurtleSim robot, determined by the `simulate` instruction. In these examples, until explicitly stated otherwise, a Kobuki will be used, that offers a more realistic robotic interface. The types and

[8] `http://rosy.inesctec.pt`

associated fields used in the examples will therefore spell the standard Kobuki ROS message types. Still, the astute reader may fittingly ask "for how long are we telling the robot to move forward?" Being ROSY a reactive programming language, the answer is not "once" or "for a certain period of time", but actually "forever". The intuition is that a controller is a function that unceasingly listens for inputs, and for each received input produces an output, what grants ROSY programs an implicit notion of time. Here `move` receives no inputs, so it will produce `Velocity` outputs at a fixed rate.

To make things a bit more interesting, imagine that we want the robot to accelerate forward with non-constant velocity. We can achieve this behavior by making sure to increase the robot's velocity at each point in time:

Example 2 (Accelerate forward).

```
accelerate :: Velocity -> Velocity
accelerate (Velocity vl va) = Velocity (vl+0.5) va
```

Note that the same `Velocity` type has different input and output meanings. This second `accelerate` controller is a function that repeatedly asks the robot for its current linear velocity, and commands the robot to increase it by 0.5 m/s.

As our robot is moving forward, what if it hits a wall? We can naturally express multiple ROSY controllers that react to distinct events. For instance, we can make the robot play an error sound when one of its bumpers is pressed, what will happen on contact with a wall:

Example 3 (Accelerate and play a sound on collision).

```
play :: Bumper -> Maybe Sound
play (Bumper _ Pressed)  = Just ErrorSound
play (Bumper _ Released) = Nothing

accelerateAndPlay = (accelerate,play)
```

In this example, we define a composite `accelerateAndPlay` controller by simply pairing together `accelerate` and `play`. Note that these two functions are not required to execute at the same time: `accelerate` runs on periodic robot information, though `play` waits for `Bumper` events. The two controllers will execute in parallel, effectively combining both behaviors. To be able to play a sound only when a bumper is `Pressed`, and not `Released`, the `play` function may or may not produce a `Sound` command. The `play` ROSY controller displays the same behavior as the ROS one encoded in Snippet 1, but here its reactive nature is clear from the type declaration.

Even though the controller from Example 3 detects when the robot hits a wall, it will continue to push against the wall, likely reaching a deadlock. With controllers as pure functions that react to events as they happen to be, there is no easy way to make a controller remember some event in the past. This contrasts with traditional imperative robot programming languages, where we could use a global variable to, e.g., memorize when the robot has hit a wall and, from then

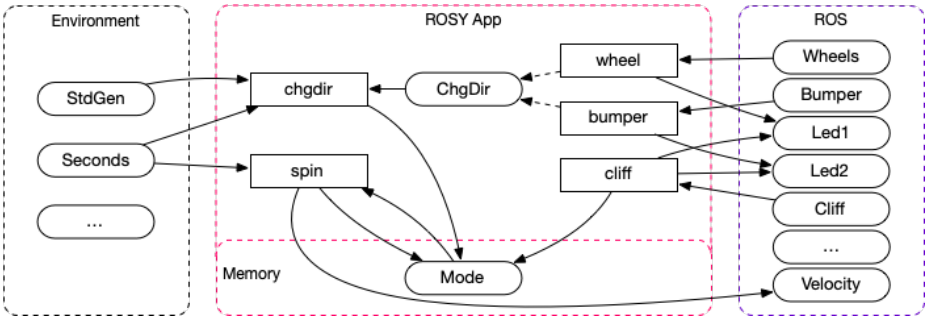


Fig. 2: The random walker application in ROSY.

on, change its behavior. Like other pedagogical functional languages [12], we grant ROSY controllers a notion of `Memory`, that can be used to, e.g., remember the robot's moving direction:

Example 4 (Accelerate forward, then backwards on collision).

```
type Hit = Bool

reverseDir :: Bumper -> Memory Hit
reverseDir _ = Memory True

accelerate :: Memory Hit -> Velocity -> Velocity
accelerate (Memory hit) (Velocity vl va) =
    if hit then Velocity (vl-0.5) va else Velocity (vl+0.5) va

forwardBackward = (reverseDir,accelerate)
```

Here the controller hands over its memory to `accelerate`, that uses it to determine in which direction to move. To reconcile memory with pure functional programming, controllers that may change the memory must return it explicitly as an output. The `Hit` boolean memory will be false by default, and set to true by `reverseDir` when a bumper event occurs.

3.2 Revisiting ROS controllers

For a more complete and realistic example, we now encode the popular Kobuki random walker controller in ROSY, while trying to stay faithful to the C++ ROS implementation[9]:

- On a bumper or cliff event, the robot blinks one of its LEDs orange and decides to change its direction;
- On a wheel drop event, the robot blinks both LEDs red and decides to stop moving while the wheel is in the air;
- When changing direction, it randomly decides on an angle between 0° and 180° and on a left/right direction. Depending on a fixed angular velocity, it estimates how many seconds it shall turn;

[9] `https://github.com/yujinrobot/kobuki/tree/devel/kobuki_random_walker`

– A sequential loop routinely performs the adequate action depending on the state of the controller. It commands the robot to either: go forward, stop moving, or turn in a given direction for a number of seconds.

Example 5 (Random walker).

```
data Mode = Go | Stop | Turn Double Seconds
data ChgDir = ChgDir -- change direction

vel_lin = 0.5
vel_ang = 0.1

bumper :: Bumper -> (Led1,Maybe ChgDir)
bumper (Bumper _ st) = case st of
  Pressed -> (Led1 Orange,Just ChgDir)
  Released -> (Led1 Black,Nothing)

cliff :: Cliff -> (Led2,Maybe ChgDir)
cliff (Cliff _ st) = case st of
  Hole -> (Led2 Orange,Just ChgDir)
  Floor -> (Led2 Black,Nothing)

wheel :: Wheel -> (Led1,Led2,Memory Mode)
wheel (Wheel _ st) = case st of
  Air -> (Led1 Red,Led2 Red,Memory Stop)
  Ground -> (Led1 Black,Led2 Black,Memory Go)

chgdir :: ChgDir -> StdGen -> Seconds -> Memory Mode
chgdir _ r now = Memory (Turn (if b then 1 else -1) time)
    where (b,r') = random r
          (ang,_) = randomR (0,pi) r'
          time = now + doubleToSeconds (ang/vel_ang)

spin :: Memory Mode -> Seconds -> (Velocity,Memory Mode)
spin m@(Memory Stop) _ = (Velocity 0 0,m)
spin m@(Memory (Turn dir t)) now | t > now = (Velocity 0 (dir*vel_ang),m)
spin m _ = (Velocity vel_lin 0,Memory Go)

randomWalk = (bumper,cliff,wheel,chgdir,spin)
```

The `randomWalk` controller encodes each part of the above specification as a separate function, sharing a memory `Mode` that encodes the different robot states. The ROS-style computation graph is illustrated in Fig. 2: nodes are defined reactive functions, and topics are user-defined or Kobuki events. The `bumper` and `cliff` functions change LED colors, and set in motion a change in direction. For greater modularity, they both emit a new event of type `ChgDir`. The `wheel` function also changes the LEDs' colors and sets the memory mode to the `Stop` state. The `spin` function reads the memory mode, sets the respective velocity depending on the mode, and returns an updated mode. It receives the current time in `Seconds` to determine if the estimated turning time has elapsed.

The most complicated behavior is left to function `chgdir`, that resolves `ChgDir` events to concrete `Turn` actions. To implement random behavior, it resorts to a Haskell standard randomness generator of type `StdGen` to generate a random direction and angle, and reads the current time to calculate the time limit for the `Turn` action. This is a good example of how the power of the full Haskell language can be gradually unleashed as students tackle more advanced problems.

Another popular Kobuki controller is the safety controller[10], that imposes stricter conditions on dangerous events. Its C++ ROS implementation can be encoded in ROSY as a single controller that reacts to bumper, cliff or wheel drop events and cautiously decides on a new velocity to escape danger:

Example 6 (Safety controller).

```
safetyControl :: Either (Either Bumper Cliff) Wheel -> Maybe Velocity
safetyControl = ...
```

The code for `safetyControl` is conceptually simple, yet verbose as it explores multiple combinations of sensor inputs. We omit it in the paper, but it can be found at the ROSY website.

The safety controller does not do much by itself, and it is typically deployed together with the random walker to limit its actions as the robot roams around. Since both controllers publish possibly conflicting `Velocity` commands to the robot, the traditional ROS solution is to use a multiplexer that remaps topics and allows one controller at a time to command the robot, according to a fixed set of priorities.

We can define a general multiplexer in ROSY as follows:

Example 7 (Binary multiplexer, `M1` with priority over `M2`).

```
data M = Start | Ignore Seconds
data M1 a = M1 a
data M2 a = M2 a

mux :: Seconds -> Memory M -> Either (M1 a) (M2 a) -> Maybe (a,Memory M)
mux t _ (Left (M1 a)) = Just (a,Memory (Ignore(t+d)))
mux t (Memory (Ignore s)) _ | s > t = Nothing
mux t _ (Right (M2 a)) = Just (a,Memory Start)
```

This binary multiplexer reads the current time `t` in `Seconds` and reacts to events either marked as `M1` or `M2`, giving higher priority to `M1` events by setting a time interval from `t` to `t+d` (for a fixed duration `d`) during which all `M2` events are ignored.

We can then instantiate a safe random walker by remapping output velocities of the safety and random walker controllers with `M1` and `M2` tags and running a multiplexer in parallel:

Example 8 (Random walker with safety controller).

[10] `https://github.com/yujinrobot/kobuki/tree/devel/kobuki_safety_controller`

```
safetyControl :: ... -> Maybe (M1 Velocity)
spin :: ... -> (M2 Velocity,...)
muxVel :: ... -> Either (M1 Velocity) (M2 Velocity) -> Maybe (Velocity,...)

safeRandomWalk = (randomWalk,safetyControl,muxVel)
```

The complete refactored code for the `safeRandomWalk` controller is available at the ROSY website.

3.3 Revisiting block-based languages

Because controllers in ROSY run forever, they do not directly lend themselves to performing sequences of instructions. For concreteness, imagine that we want to command the robot to draw a square on the floor with its movement. In a visual robot programming language, this can be done by assembling a block like the one from Fig. 1. Even though such a block can be expressed in ROSY as a multi-stage controller that explicitly encodes a state machine and reacts differently depending on the state[11], it is useful to lend more structure to the language as the complexity of the controller grows.

Like other FRP languages [22], ROSY introduces the concept of tasks, as a continuous controller and a terminating event. For example, we can make the robot turn sideways by a fixed amount of degrees by writing a task:

Example 9 (Task: Turn left or right).

```
type Side = Either Degrees Degrees

turn :: Side -> Task () ()
turn s = task runTurn (taskOpts { init = startTurn s })

startTurn :: Side -> Orientation -> Memory Orientation
startTurn (Left a) o = Memory (o+degreesToOrientation a)
startTurn (Right a) o = Memory (o-degreesToOrientation a)

runTurn :: Memory Orientation -> Orientation -> Either Velocity (Done ())
runTurn (Memory to) from = if abs d <= err
    then Right (Done ())
    else Left (Velocity 0 (orientation (normOrientation (to-from))))
```

A task receives an optional initializer that sets up the stage for the controller; in this example, the `startTurn` initializer reads the robot's `Orientation` from its odometry information, and writes the desired final `Orientation` to memory by adding or subtracting the received angle to the current orientation. The `runTurn` controller will rotate the robot towards the desired orientation until the desired and current orientations are equal with a small error margin `err`, signalling when it is `Done`. The `Done` type allows returning additional information on task termination. Returning nothing is achieved with the empty type `()`.

[11] Classical FRP frameworks tackle this general problem by designing advanced higher-order switching combinators over reactive functions, that are expressively powerful but even less novice-friendly.

A similar task makes the robot move a fixed distance:

Example 10 (Task: Move forward or backwards).

```
data Direction = Forward  Centimeters | Backward Centimeters

move :: Direction -> Task () ()
move = ...
```

Since tasks can end, in contrast to controllers, we can now mimic the block from Fig. 1 in ROSY by using Haskell's monadic notation to sequence tasks in an imperative style:

Example 11 (Task: Draw a square).

```
drawSquare :: Task () ()
drawSquare = replicateM_ 4 $ do { move (Forward 2); turn (Left 90) }
```

3.4 Supporting services and multiple robots

In ROSY (as in Haskell), the monadic syntax allows a crisp distinction between the continuous (functional) world of controllers and the synchronous (imperative) world of tasks, that promote two different styles of composition: multiple controllers interact with the robot in parallel, while multiple tasks execute in sequence. Aside from deployment details, a ROS service can be seen as a client-side function: a client sends a request and waits for a response. Since request and response are not ROS events, and a service caller blocks until it gets a response, services are best modelled as ROSY task-level functions.

As customary in ROS tutorials, we will use the TurtleSim to exemplify the usage of services in ROSY. In terms of controllers, a TurtleSim turtle is a simpler version of a Kobubi that only supports three events for `Velocity`, `Position` and `Orientation`. These are distinguished from the similarly-named Kobuki events by indicating a fixed event-level `TurtleNumber`, which is just an `Int` that identifies each `Turtle`. For instance, we can easily adapt one of our ROSY examples to make the first turtle accelerate:

Example 12 (Accelerate the first turtle).

```
accelerate :: Turtle 1 Velocity -> Turtle 1 Velocity
accelerate (Turtle (Velocity vl va)) = Turtle (Velocity (vl+0.5) va)

main = simulate Turtlesim accelerate
```

We change the argument to `simulate` accordingly. From now on, our examples will assume a TurtleSim.

Adapted to turtle events, we can also reuse the previous logic of the `turn` and `move` tasks to control a specific turtle:

Example 13 (Task: Turn and move turtle).

```
turn :: TurtleNumber -> Side -> Task () ()
move :: TurtleNumber -> Direction -> Task () ()
```

Besides events regarding the state of each turtle, the TurtleSim offers a few simple services for managing the number of turtles and controlling the properties of each turtle's position and pen, encoded as task-level functions in ROSY:

```
spawn :: Position -> Orientation -> Task () TurtleNumber
kill :: TurtleNumber -> Task () ()
setPen :: TurtleNumber -> Pen -> Task () ()
teleportAbsolute :: TurtleNumber -> Position -> Orientation -> Task () ()
...
```

The `spawn` service spawns a new turtle with the provided parameters, and returns its number; the other listed services return an empty response, but their invoking task (or controller as we will see later) will nonetheless wait for them to complete.

We can now seamlessly combine tasks and services to make a turtle draw an endless spiral:

Example 14 (Task: Draw a colored spiral).

```
spiral :: TurtleNumber -> Double -> Double -> Double
       -> Color -> (Color -> Color) -> Task () ()
spiral n len ang width c upd = do {
    setPen n (Pen c (floor width) On);
    move n (Forward len); turn n (Left ang);
    spiral (len+0.02) (ang-0.5) (width+0.2) (upd c) upd }

red_spiral1 = spiral 1 0.2 30 1 black tored
    where tored (Color r g b) = Color (r+10) g b
```

The `spiral` parametrized task is a recursive function that interpolates the shape of a spiral by having the turtle moving forward and turning left by increasing distance and decreasing angle. The `setPen` service is used in each step to increase the thickness of the pen and lighten the color, where a `Color` is given by its (r,g,b) components between 0 and 255; `red_spiral1` is the top-level task to be simulated, and draws a red gradient spiral with the first default turtle.

Sometimes it is useful to write controllers that may dynamically control more than one turtle, depending on particular events. For instance, we can write a controller that makes all active turtles follow a designated leader.

Example 15 (Task: Follow the leader).

```
advertise :: Param TurtleNumber -> AnyTurtle Position
          -> Maybe (Memory Position)
advertise (Param n1) (AnyTurtle n2 p) = if n1 == n2
  then Just (Memory p) else Nothing

follow :: Param TurtleNumber -> Memory Position -> AnyTurtle Pose
       -> (AnyTurtle Velocity)
follow (Param l) (Memory p2) (AnyTurtle n pose)
  | n == l = AnyTurtle n (Velocity 0.5 0.2)
  | otherwise = AnyTurtle n (Velocity vl va)
      where p1 = posePosition pose
```

```
    Orientation o = poseOrientation pose
    vec = subVec (positionToVec p2) (positionToVec p1)
    a = angleVec vec
    vl = if abs va < 0.2 then magnitudeVec vec else 0
    va = normRadians (a - o)

followTheLeader :: Task () ()
followTheLeader = task (advertise,follow) taskOpts
```

In this example the `TurtleNumber` of the leader is defined via a global `Parameter`, that may be read/set in parallel at any time by any other controller or task. The `followTheLeader` task uses task-local `Memory` to record the current `Position` of the leader. The `advertise` controller listens to all turtle's positions and advertises new `Position`s of the leader. The `follow` controller makes all other turtles but the leader move towards the most recently recorded leader's `Position`. The special parameterized `AnyTurtle` event allows subscribing/publishing an event for all the active turtles.

3.5 Integrating actions

So far, we have seen how the task notation greatly simplifies the design of sequential robotic tasks, which comes precisely from the fact that the controller waits for the result of the task. However, within the ROS ecosystem, the synchronous nature of services is fit for short tasks (such such as changing the pen style), but ill-advised for long-running tasks (such as drawing a spiral), since it prevents the calling controller from doing anything in-between, including preempting the task itself. For long-running tasks, the recommended ROS interface is that of asynchronous actions: a client can call an action with a request and register a callback through which it will get a response once the task terminates (and, possibly, progress feedback in the meantime). Since the main difference is that a calling controller does not block waiting for a response, we model actions in ROSY by calling a task from within a continuous, asynchronous controller. This is similar to the ROS approach to actions, which is a library also built on top of the primitive ROS communication mechanisms (namely topics and services).

Imagine, for instance, that we want to spawn a second turtle and have two turtles drawing spirals of different angles and colors. With the mechanisms presented thus far, such tasks would have be sequential, but ROSY also supports wrapping them as asynchronous tasks.

Example 16 (Task: Draw two colored spirals).

```
green_spiral2 = do {
    turtle <- spawn turtlesimDefaultPosistion turtlesimDefaultOrientation;
    spiral turtle 0.3 40 1 black togreen }
  where togreen (Color r g b) = Color r (g+10) b

spiral12 = (call red_spiral1 callOpts,call green_spiral2 callOpts)
```

The `green_spiral2` task spawns a second turtle in the same position as the first one and then, similarly to `red_spiral1`, makes it draw a green spiral with a

narrower angle. A function `call` creates a new context within a controller in which it executes a synchronous task; it returns an event that issues the call and may wait for the result of the task by passing additional parameters to `callOpts`. The two spiral tasks are put together by the top-level `spiral12` controller.

Given its asynchronous nature, a ROS action supports additional features beyond services. Besides the final response, an action may provide frequent feedback to the caller during its execution, and the caller may cancel the action before it terminates. To illustrate how these concepts can be modeled using ROSY tasks, consider the following task that mimics the `rotateAbsolute` action offered by the TurtleSim since ROS 2 Foxy[12] and makes a turtle rotate to an absolute orientation:

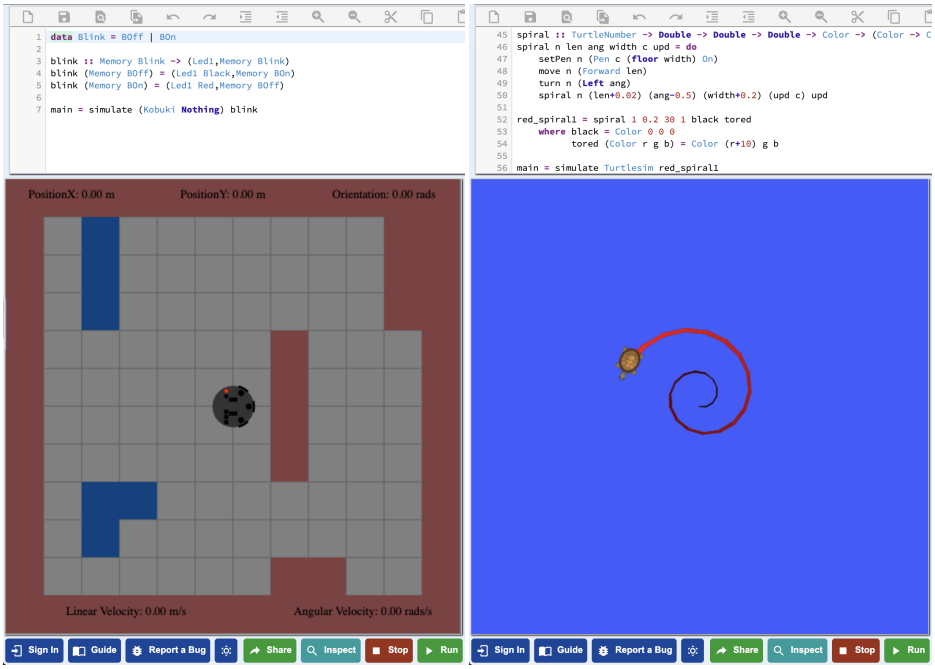
Example 17 (Task: Rotate turtle to absolute orientation, action server).

```
startRotAbs :: Turtle n ()
            -> Turtle n Orientation -> Seconds -> (Memory Radians)
startRotAbs _ (Turtle (Orientation o)) now = (Memory o)

doRotAbs :: Radians -> Turtle n () -> Turtle n Orientation
  -> Memory Radians -> Either (Feedback Radians,Turtle n Velocity)
                              (Turtle n Velocity,Done Radians)
doRotAbs dest _ (Turtle (Orientation now)) (Memory start) =
    if abs remaining <= 0.02
        then Right (Turtle (Velocity 0 0),Done delta)
        else Left (Feedback remaining,Turtle (Velocity 0 vl))
  where delta = normRadians (start - now)
        remaining = normRadians (dest - now)
        vl = if remaining < 0 then -remaining else remaining

cancelRotAbs :: Turtle n () -> (Turtle n Velocity)
cancelRotAbs _ = (Turtle (Velocity 0 0))

rotateAbsolute :: TurtleNumber -> Radians -> Task Radians Radians
rotateAbsolute n ang = onTurtle n (\t -> task (doRotAbs ang t) (opts t))
    where opts t = TaskOpts (startRotAbs t) (cancelRotAbs t)
```

Here task `rotateAbsolute` encodes the action server. It receives the desired orientation, and its initializer event calculates and memorizes the angular displacement to the starting position. In each step, `doRotAbs` either instructs the turtle to rotate and reports the `remaining` distance as `Feedback`, or identifies the task as finished and returns `Done`. If the task is cancelled, it sets the velocity of the turtle to zero to make the turtle stop before exiting (since the TurtleSim remembers the last issued `Velocity` command for the duration of one second). The `onTurtle` operator converts a value-level turtle number to an event-level turtle number, which is used to connect the controllers to the appropriate turtle.

Example 18 (Task: Rotate turtle to absolute orientation, action client).

[12] `https://github.com/ros/ros_tutorials/tree/foxy-devel/turtlesim`



(a) Kobuki simulator (b) TurtleSim simulator

Fig. 3: The CodeWorld-powered ROSY environment.

```
cancel_halfway :: Turtle 1 Orientation -> Maybe Cancel
cancel_halfway (Turtle (Orientation o)) =
    if o >= pi/2 then Just Cancel else Nothing

cancel_rotate1 = call (rotateAbsolute 1 pi) opts
    where say_feed f = Say ("feedback " ++ show f)
          say_done d = Say ("done " ++ show d)
          opts = CallOpts cancel_halfway say_feed say_done
```

The `cancel_rotate1` controller acts as a (non-blocking) action client. It calls the `rotateAbsolute` action to rotate the first turtle for π radians, and cancel the action halfway. The additional options passed to `call` specify how to react to received feedback and end events, in this case by printing to the console.

4 Environment

As well as striving to allow students to learn a "real" programming language while freeing them from inessential technical language details outside of how to control a robot, ROSY comes with a fully integrated development environment. To really allow students to concentrate on the meaning of their programs, ignoring deployment details, the ROSY environment runs as a web application inside any modern web browser.

Fig. 3 shows the ROSY environment in action, both with the Kobuki and the TurtleSim simulators running some of the examples from Section 3. It is pow-

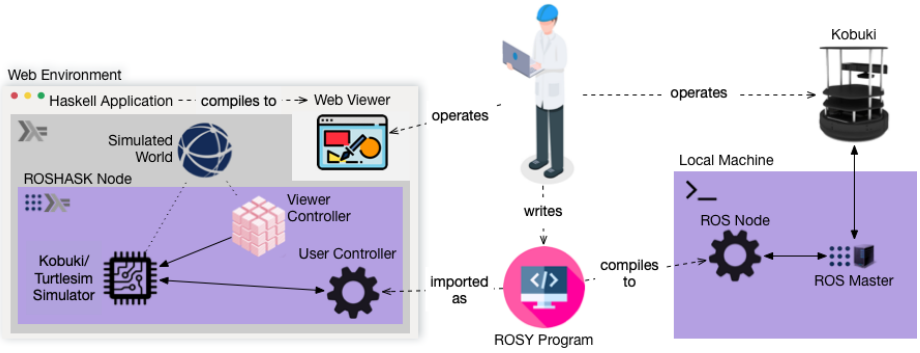


Fig. 4: The ROSY architecture.

ered by Codeworld [5], a modern educational environment for writing graphical Haskell programs such as games and animations. CodeWorld has been used in K12 schools for years, and supports a code editor with features such as syntax highlighting, improved on-the-fly compiler error messages, extensive documentation or easy sharing of projects.

Another vital component of the environment is a visualizer that simulates, directly in the browser, how the controller programmed by the student interacts with a robot in a fictitious 2D world. At the moment, this is tailored for a TurtleBot2 placed in a tiled world made of floor, walls and cliffs. In ROSY training sessions for K12 students that we have hosted at the University of Minho, students could also deploy their same code to control a real robot and perceive the differences between a simulated and a real world. Enabling students to simultaneously test their examples in simulated and real scenarios played an important role in teaching them the importance of these differences and their influence on the design of approximate, event-driven robotic controllers.

5 Under the hood

Despite their simplicity, ROSY programs are fully compatible with the ROS infrastructure. Under the hood, the ROSY language is implemented as an embedded domain-specific language that provides an additional abstraction layer over the existing Haskell ROS library. The ROSY environment is also implemented in Haskell, by extending and specially tailoring the CodeWorld [5] environment to ROSY. This includes a custom prelude that is imported by default, a custom code pre-processor to automatically derive necessary Haskell type class instances, and a custom graphical simulation of the robot. All the source code is open-source and freely available[13].

5.1 Haskell ROS library

The ROSHASK library [8] enables ROS programming within the Haskell ecosystem by supporting the deployment of ROS client nodes that are compatible

[13] `https://github.com/hpacheco/codeworld-rosy`

with the `TCPROS` communications protocol[14]. ROSHASK fades the architectural boundaries between ROS system and software components, by lifting topics to first-class values and designing a collection of FRP-style combinators to split, fuse and generally manipulate topics in a more expressive, modular and compositional way.

At its core, the library provides functions for subscribing and publishing topics:

```
subscribe :: String -> Node (Topic m a)
publish   :: String -> Topic m a -> Node ()
```

The type variable `m` is a monad for actions with effects, typically `IO` for interacting with a non-pure outside world, and `a` is the type of the subscribed or published event derived from standard ROS message type definition files. Topics are modeled as infinite monadic streams [21], i.e., monadic actions that produce the next value and a new topic:

```
newtype Topic m a = Topic (m (a,Topic m a))
```

The `Node` monad manages TCP connections and internal buffers of published and subscribed messages. The following example node subscribes to two sensors, fuses them using the `bothNew` combinator (subsampling the faster topic), maps an action `act :: (Sense1,Sense2) -> Cmd` over each pair of sensor values, and publishes the resulting commands:

```
n1 = do { t1 <- subscribe "sense1"; t2 <- subscribe "sense2"
        ; publish "cmd" $ fmap act $ t1 `bothNew` t2 }
```

Internally, the asynchronous ROSHASK behavior is implemented on top of Haskell user-space threads, that are managed by the Haskell runtime and much more efficient than system threads. For instance, a typical `publish` implementation launches a thread that infinitely samples values from a topic and communicates them to the ROS master; similarly, merging two topics is done by launching two threads that independently consume each topic and write to a common channel.

Nodes can also be easily composed, for instance, we can simultaneously install a handler that listens to and prints the published commands to the command line:

```
n2 = subscribe "cmd" >>= runHandler putStrLn
```

In the style of `publish`, `runHandler` is a general combinator that launches a thread that will consume values from a topic and execute some user-defined `IO` action:

```
runHandler :: (a -> IO b) -> Topic IO a -> Node ()
```

The composite node `n1 >> n2` will communicate the commands produced by `n1` both via `rostcp` and locally to `n2`.

[14] ROSHASK topics are independent of the communication protocol. To support, e.g., ROS 2, suitable client library bindings may be required.

5.2 Architecture

The ROSY web environment, depicted on left side of Fig. 4, is designed according to the common Model–View–Controller pattern, with components implemented as separate ROSHASK controllers that communicate locally and form a single ROSHASK node. The model controller (loosely) simulates a standard `kobuki_node`[15] or `turtlesim`[16] in a simulated world, the view controller implements a simple 2D animation of the robot within the world, and the user controller represents the node specified by the programmer. To support web-based simulation and loosen the dependency on ROS, we have adapted the ROSHASK library to run without a ROS master server. This way, all the Haskell code needed to perform a simulation is compiled via GHCJS into a JavaScript application that runs directly in the client browser.

Alternatively to web-based simulation, ROSY programs can also be executed on a local machine (right side of Fig. 4), by connecting to a ROS master server and operate a more realistic Gazebo simulator or a real Kobuki robot. We have tested both scenarios on Ubuntu 14.04 with ROS Indigo and a TurtleBot2. Currently, the TurtleSim interface is only supported via web-based simulation, and we have not tested a simulated ROS TurtleSim node. Although ROSY tasks are designed to be fully compatible with ROS services and actions, they currently only run locally within our simulated ROSHASK web backend; registering them with a ROS master would require additional ROSY constructs to address namespacing and communication details and also extending the existing ROSHASK support for services and actions, e.g., to support registering new services and actions.

5.3 Implicit stream programming

The greatest design decision of the ROSY language is that controllers have an implicit notion of time. This favors a simpler declarative style focused on *which* commands are issued *when* events happens, without specifying *how often* subscribers and publishers interact with the ROS world, and freeing programmers from some of the common robotic programming details such as clocks, sampling rates or synchronization. Therefore, unlike ROSHASK, that exposes a full-fledged API to manipulate topics as a whole, ROSY controllers are less expressive in that they only consider a single point in time.

Events are identified by their type in ROSY. Two type classes internally bind ROSY types and ROS message namespaces to streams of subscribed sensors or published commands:

```
class Sensor a where
  sensor  :: Node (Topic IO a)
class Command a where
  command :: Topic IO a -> Node ()
```

For example, the same `Velocity` type can simultaneously express the act of getting the current velocity from the robot's periodic odometry data and the act of setting the desired velocity by sending a command to the robot's base:

[15] `http://wiki.ros.org/kobuki_node`
[16] `http://wiki.ros.org/turtlesim`

```
instance Sensor Velocity where
  sensor = subscribe "odom" >>= return . fmap (_twist . _twist))
instance Command Velocity where
  command = publish "/mobile_base/commands/velocity"
```

The ROSY language currently supports a fixed set of events offered by the `kobuki_node` and `turtlesim` APIs. Extending support for other robots simply requires defining new boilerplate `Sensor` and `Command` instances, as ROSHASK already supports many standard ROS message types and allows deriving Haskell types from custom ROS message files.

To conciliate whole topics with point-wise controllers, we define another type class that implicitly lifts a function on individual values to a controller on streams of values:

```
class Controller a where
  controller :: a -> Node ()
```

The stream semantics of the lifted controller is then inferred from the function's type signature: inputs correspond to subscribed sensors, and outputs to published commands, e.g.:

```
instance (Sensor a,Command b) => Controller (a -> b) where
  controller f = sensor >>= command . fmap f
```

Instances for composite types perform implicit stream programming. For example, a `Controller` that receives an input pair is fusing data from two sensors:

```
instance (Sensor a,Sensor b) => Sensor (a,b) where
  sensor = liftM2 bothNew sensor sensor
```

As another example, a `Controller` that returns possibly different commands is splitting the output stream (using the `tee` ROSHASK combinator that duplicates a topic), and processing each type of commands independently:

```
instance (Command a,Command b) => Command (Either a b) where
  command t = do { (t1,t2) <- tee t
                 ; command (lefts t1); command (rights t2) }
```

Multiple `Controller`s are executed in parallel threads, and can be composed in sequence:

```
instance (Controller a,Controller b) => Controller (a,b) where
  controller (a,b) = controller a >> controller b
```

5.4 Supporting user-defined events, memory and parameters

ROSY also allows the declaration of user-defined data types for more modular intra-node communication. Since these need not be bound to ROS namespaces, every newly declared user-defined data type is by default a `Sensor` and a `Command`. This is supported by a new `UserNode` monad that extends `Node` capabilities with local type-indexed event buffers for user-defined data types.

Since ROSY programs are by design pure Haskell functions, there is no native support for common robotics design patterns that use global memory. It is nonetheless possible to emulate global variables by publishing an initial event, and on every update subscribing to current event and re-publishing its updated value. Even so, this pattern can be error-prone as the programmer needs to be cautious about subscribing and publishing the "variable" the right number of times in order to keep it alive. This may also be problematic if more than one controller is manipulating the same "variable" in parallel.

To avoid these caveats, ROSY supports task-local memory (if a program has no tasks, then memory is global) and global parameters (akin to ROS parameters). We extend the `UserNode` monad with two transactional memory stores, holding a global value for every distinct type `a`, distinguishable through two type-level tags:

```
data Memory a = Memory a
data Param a = Param a
```

In order to support transactional controllers, or more specifically, be able to execute each controller thread as a single transaction, we must generalize our sensor and command interfaces to produce and consume topics of transactions[17]:

```
sensor   :: UserNode (Topic IO (STM a))
command  :: Topic IO (STM (Maybe a)) -> UserNode (Topic IO (STM ()))
```

A `Sensor (Memory a)` returns a topic that repeatedly reads from the memory variable of type `a`, while a `Command (Memory a)` appends memory writes to a topic of transactions, returning a new topic; `Param` behaves similarly. The greatest change occurs on commands that, unlike before, must be published synchronously within the same transaction, meaning we can no longer fork a topic and publish each side independently. We can execute a controller thread by installing a handler that `atomically` executes each transaction as a side effect:

```
instance (...) => Controller (a -> b) where
  controller f = sensor >>= command . fmap f
               >>= lift . runHandler atomically
```

5.5 Supporting multiple robots

The TurtleSim supports 9 different turtles, with topics identified by their number, e.g., each turtle `X` publishes its status to `"/turtleX/pose"`. As ROS topics map to ROSY types, we create a type-level tag `Turtle n a` that identifies the event `a` for turtle `n`. Here, `n` is a type-level number from 1 to 9. As in some examples that we have seen above, it is possible to convert a value-level turtle number to a type-level turtle number, in order to define turtle-generic tasks/controllers using the `onTurtle :: TurtleNumber -> (forall n . Turtle n () -> b) -> b` operator as long as the behavior of the task/controller is not bound to a concrete turtle (which is precisely captured by the type signature).

[17] The `STM` monad stands for Haskell's software transactional memory library. The `Maybe` type is a technical requirement for filtering values inside a transaction, since instances must not change the periodicity of the topics.

In more advanced examples, it may be useful to dynamically control turtles whose numbers are only known at run-time. For such cases, we offer a `AnyTurtle a` type-level tag pairs an event `a` with a value-level `TurtleNumber`; its `Sensor` instance subscribes to the events of all nine turtles and its `Command` instance publishes to the topic of any turtle depending on the event's value.

5.6 Supporting tasks

In ROSHASK and other FRP approaches, topics are modeled as infinite streams and topic handlers are program-long threads continuously waiting on and reacting to events. The fact that the data flow graph, inferred from the wiring of stream combinators, is typically known statically, allows ROSHASK to register all subscribers and publishers with the ROS master at node initialization, before starting to actually process data.

In ROSY, a task is defined as a continuous controller and a terminating event:

```
task :: (Command init,Command cleanup,Controller ctrl)
     => ctrl -> TaskOpts init cleanup -> Task feedback end
```

It also receives as optional arguments an initialization step that is executed once at startup and a cleanup step that is executed once before exiting when it is cancelled; default options with no initialization and no cleanup are provided by `taskOpts`. A task controller issues termination via a special event type:

```
data Done a = Done a
```

We also make `Task` a monad, so that programmers can use monadic notation to sequence tasks. For composing two smaller tasks in into a composite task, where the output of the first is passed on to the second, we may write:

```
task12 = do { end1 <- task init1 ctrl1; task (init2 end1) ctrl2 }
```

In this scenario, controllers no longer run forever: when the first task ends, we must uninstall the controller `ctrl1` and install a new controller `ctrl2` for the second task. We have extended ROSHASK to support dynamic node configuration: each task runs within its own `UserNode`; publishers and subscribers are registered at declaration time; tasks keep a fine-grained control of launched threads, and all children threads are killed when exiting the parent `UserNode`[18]. `Memory` is local to each `UserNode`, while a `Param` lives through the whole `Node`.

We offer bindings to existing ROS services as tasks via ROSHASK, e.g., for killing a turtle:

```
kill :: TurtleNumber -> Task () ()
kill i = nodeTask (callService "kill" (KillRequest ("turtle"++show i)))
    >>= \e -> case e of { Left err -> return ()
                        ; Right KillResponse -> return () }
```

[18] Note that messages are not lost when transitioning between tasks, since a `Node` keeps global buffers of published and subscribed ROS topics.

The request and response types are automatically derived by ROSHASK from ROS service description files. Here, `nodeTask` embeds a `Node` within a `Task`. We use default turtle names from their numbers and ignore service invocation errors.

Analogously to ROS actions, a `Task` can additionally be cancelled or provide feedback during its execution. These features become evident when calling a task from a controller:

```
call :: (Sensor when,Command see,Command res) => Task feedback end
     -> CallOpts when feedback see end res -> Call
```

In order to conciliate tasks with the continuous nature of controllers, a `call` produces a special `Call` event whose `Command` instance actually calls the task and registers the necessary callbacks. Besides the called task, it receives the following optional arguments (with sensible defaults in `callOpts`):

- A callback that can read any sensor to decide `when` to send a special `Cancel` event to the abort the task; in such a case, the task does not produce its terminating event `end`. Since tasks can themselves be a composition of smaller tasks, when a composite task is cancelled the `Cancel` event is automatically propagated to its sub-tasks.
- A callback that reads all `feedback` produced by the task, and issues a command that allows the controller to `see` part of it. Note that `feedback` is also an argument of the `Task` type, and therefore must be the same for all subtasks of a composite task. Task feedback can be changed using the operator `subTask :: (f1 -> Maybe f2) -> Task f1 a -> Task f2 a`.
- A callback that takes the terminating event `end` and publishes it back to the controller as a command `res`.

Combining tasks and controllers, we can readily define other high-level asynchronous programming combinators in the style of the Haskell `async` library[19], e.g., to run two tasks concurrent in parallel and wait for both to terminate (returning a pair) or cancel the other when one terminates (returning an option):

```
data PDone a = PDone { unPDone :: a }

lCallOpts = callOpts { feedback = Feedback . Left, response = PDone }
rCallOpts = callOpts { feedback = Feedback . Right, response = PDone }

parallel :: Task f1 a -> Task f2 b -> Task (Either f1 f2) (a,b)
parallel t1 t2 = task pdone (taskOpts { init = (c1,c2) })
    where c1 = call t1 lCallOpts
          c2 = call t2 rCallOpts
          pdone (PDone a) (PDone b) = Done (a,b)

race :: Task f1 a -> Task f2 b -> Task (Either f1 f2) (Either a b)
race t1 t2 = task rdone (taskOpts { init = (c1,c2) })
    where c1 = call t1 lCallOpts
          c2 = call t2 rCallOpts
          rdone = (Done . Left . unPDone,Done . Right . unPDone)
```

[19] https://hackage.haskell.org/package/async

6 Conclusion

In this paper we have presented ROSY, a new pedagogical robot programming language that advocates a sweet-spot between the expressiveness of FRP and the needed simplicity of an educational setting. As part of a computing summer camp held at the University of Minho[20], in July 2019, we have taught a 4-hour training session for K12 students on hands-on robot programming in ROSY, where students are asked to program the robot to perform a series of simple tasks in the style of our first four examples, with only two prior sessions of general programming in Haskell. From our perceptions, students were able to comprehend the concepts, quickly start programming and execute different tasks.

In the future, we plan to provide further ROSY training sessions and undergo empirical studies to corroborate its practical value for learning programming via robotics. We plan to improve the ROSY environment with more advanced simulation scenarios using, e.g., the Gazebo web client[21] or remote ROS support similar to [6, 24]. Orthogonally, we intend to migrate ROSY and ROSHASK to ROS 2 in order to explore the more modular support for ROS nodes composition.

We also plan to explore the design of novel novice-friendly interfaces that blend textual and visual representations, including blocks, state machines or data flow diagrams. To fulfill the appeal of declarative robot programming beyond its pedagogical use, it would also be interesting to explore how students can transition from ROSY to ROSHASK or other full-fledged FRP frameworks. On that account, we are also interested in conducting more fundamental FRP research on unifying ROSHASK with mainstream FRP frameworks.

References

1. Almeida, J., Cunha, A., Macedo, N., Pacheco, H., Proença, J.: Teaching how to program using automated assessment and functional Glossy games (experience report). PACMPL **2**(ICFP), 82:1–82:17 (2018)
2. Angulo, I., García-Zubía, J., Hernández-Jayo, U., Uriarte, I., Rodríguez-Gil, L., Orduña, P., Pieper, G.M.: RoboBlock: A remote lab for robotics and visual programming. In: exp.at. pp. 109–110. IEEE (2017)
3. Bainomugisha, E., Carreton, A.L., Cutsem, T.V., Mostinckx, S., Meuter, W.D.: A survey on reactive programming. ACM Comput. Surv. **45**(4), 52:1–52:34 (2013)
4. Blank, D.S.: Robots make computer science personal. Commun. ACM **49**(12), 25–27 (2006)
5. Breitner, J., Smith, C.: Lock-step simulation is child's play (experience report). PACMPL **1**(ICFP), 3:1–3:15 (2017)
6. Casañ, G.A., Cervera, E., Moughlbay, A.A., Alemany, J., Martinet, P.: ROS-based online robot programming for remote education and training. In: ICRA. pp. 6101–6106. IEEE (2015)
7. Cleary, A., Vandenbergh, L., Peterson, J.: Reactive game engine programming for STEM outreach. In: SIGCSE. pp. 628–632. ACM (2015)
8. Cowley, A., Taylor, C.J.: Stream-oriented robotics programming: The design of roshask. In: IROS. pp. 1048–1054. IEEE (2011)

[20] `https://www.uminho.pt/veraonocampus`
[21] `http://gazebosim.org/gzweb.html`

9. Diprose, J.P., MacDonald, B.A., Hosking, J.G.: Ruru: A spatial and interactive visual programming language for novice robot programming. In: VL/HCC. pp. 25–32. IEEE (2011)
10. Druin, A., Hendler, J.A., Hendler, J.: Robots for kids: Exploring new technologies for learning. Morgan Kaufmann (2000)
11. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: The TeachScheme! project: Computing and programming for every student. Computer Science Education **14**(1), 55–77 (2004)
12. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: A functional I/O system or, fun for freshman kids. In: ICFP. pp. 47–58. ACM (2009)
13. Gandy, E.A., Bradley, S., Arnold-Brookes, D., Allen, N.R.: The use of LEGO Mindstorms NXT robots in the teaching of introductory Java programming to undergraduate students. ITALICS **9**(1), 2–9 (2010)
14. Hudak, P., Courtney, A., Nilsson, H., Peterson, J.: Arrows, robots, and functional reactive programming. In: Advanced Functional Programming. LNCS, vol. 2638, pp. 159–187. Springer (2002)
15. Lawhead, P.B., Duncan, M.E., Bland, C.G., Goldweber, M., Schep, M., Barnes, D.J., Hollingsworth, R.G.: A road map for teaching introductory programming using LEGO Mindstorms robots. SIGCSE Bulletin **35**(2), 191–201 (2003)
16. Marghitu, D., Coy, S.: Robotics rule-based formalism to specify behaviors in a visual programming environment. In: B&B@VL/HCC. pp. 45–47. IEEE (2015)
17. Masum, M.H., Rifat, T.S., Tareeq, S.M., Heickal, H.: A framework for developing graphically programmable low-cost robotics kit for classroom education. In: ICETC. pp. 22–26. ACM (2018)
18. Oddie, A., Hazlewood, P., Blakeway, S., Whitfield, A.: Introductory problem solving and programming: Robotics versus traditional approaches. ITALICS **9**(2), 1–11 (2010)
19. Pacheco, H., Macedo, N.: ROSY: an elegant language to teach the pure reactive nature of robot programming. In: IRC. pp. 240–247. IEEE (2020)
20. Pembeci, I., Nilsson, H., Hager, G.D.: Functional reactive robotics: An exercise in principled integration of domain-specific languages. In: PPDP. pp. 168–179. ACM (2002)
21. Perez, I., Bärenz, M., Nilsson, H.: Functional reactive programming, refactored. In: Haskell. pp. 33–44. ACM (2016)
22. Peterson, J., Hudak, P., Elliott, C.: Lambda in motion: Controlling robots with haskell. In: PADL. LNCS, vol. 1551, pp. 91–105. Springer (1999)
23. Quigley, M., Conley, K., Gerkey, B.P., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: ROS: An open-source Robot Operating System. In: OSS@ICRA. vol. 3 (2009)
24. Toris, R., Kammerl, J., Lu, D.V., Lee, J., Jenkins, O.C., Osentoski, S., Wills, M., Chernova, S.: Robot web tools: Efficient messaging for cloud robotics. In: 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 4530–4537. IEEE (2015)
25. Weintrop, D.: Block-based programming in computer science education. Commun. ACM **62**(8), 22–25 (2019)
26. Weintrop, D., Afzal, A., Salac, J., Francis, P., Li, B., Shepherd, D.C., Franklin, D.: Evaluating CoBlox: A comparative study of robotics programming environments for adult novices. In: CHI. pp. 366:1–366:12. ACM (2018)
27. Xu, Z., Ritzhaupt, A.D., Tian, F., Umapathy, K.: Block-based versus text-based programming environments on novice student learning outcomes: A meta-analysis study. Computer Science Education **29**(2-3), 177–204 (2019)