Improv - Live Coding for Robot Movement Design

Alexandra Q. Nilles

February 13, 2018

Abstract

This paper introduces the Improv system, a programming language for high-level description of robot motion with immediate visualization of the resulting motion on a physical or simulated robot. The intended users of this tool are anyone looking to quickly generate robot motion, such as educators, artists, and researchers. The system includes a domain-specific language, inspired by choreographic techniques, which allows for several ways of composing and transforming movements such as reversing movements in space and time and changing their relative timing. Instructions in the Improv programming language are then executed with roshask, a Haskell client for ROS ("Robot Operating System"). ROS is an open-source robot software framework which is widely used in academia and industry, and integrated with many commercially available robots. However, the ROS interface can be difficult to learn, especially for people without technical training. This paper presents a "live coding" interface for ROS compatible with any text editor, by executing whenever the user saves changes. Currently, Improv can be used to control any robot compatible with the 'Twist' ROS message type (which sets linear and rotational velocity). This paper presents Improv implementations with the two-dimensional simulator Turtlesim, as well as three-dimensional TurtleBots in the Gazebo simulation engine.

Introduction

- what is it
- possible audiences

Related Work

One closely related project is *Improv* is *Dance*, a domain-specific language built in Haskell [1]. The project included a DSL inspired by Labanotation, as well as a reactive layer that allowed the robot to respond to sensor events. The project targeted humanoid robots, while *Improv* has so far targeted mobile robots, and *Dance* would generate the necessary 3D simulation code, though did not include the live-coding interface of this work and predates ROS. *Improv* has incorporated and adapted some of the data structures from *Dance*, namely the `Action` and `Dance` data types.

This work is also influenced by live coding interfaces and programming languages for generating music, often part of the *Algorave* performance movement. In particular, the programming language TidalCycles [2] has had a strong influence on this work, both syntactically and in how relative timing of events is managed.

Al Jazari is a live-coding installation which uses a simple graphical language to allow people to control robots (in simulation). The language includes conditionals based on external state and communication between the bots. (http://davesblog.fo.am/category/al-jazari/) The program state of the robot is also visualized.

Especially when used with the two-dimensional Turtlesim, *Improv* is reminiscent of *Logo*, an interpreted dialect of Lisp that is often used in conjunction with a simulation of a two-dimensional turtle. Our programming language has different features than *Logo*, does not support recursion as *Logo* does, and most importantly is integrated with ROS and thus able to be used with three-dimensional simulators and actual robots.

We know of two other projects have addressed the problem of the complex development cycle in ROS by creating tools for interactive or "live" programming. One such project **???** created a DSL in Python which allows for wrapping and modifying existing ROS nodes, using the Python REPL. However, by using the Python REPL, the user is only able to experiment with commands in a shell and is not able to save the commands they have tried in a file. Additionally, since the DSL is implemented as

a library in Python, it inherits some of the opaque syntax of the Python ROS client. Improv has a simpler, albeit less powerful, programming language and models movement explicitly. Another closely related work to *Improv* is the Live Robot Programming (LRP) language and its integration with PhaROS, a client library for ROS written is Pharo, a dynamic programming language specialized for live updating and hot recompilation. This project allows for live-coding of ROS nodes and reconfiguration of the ROS network with a much shorter development cycle than traditional ROS programming. However, the aims of these projects and Improv are slightly different - the DSL, while more high-level than most robot programming languages, was not designed around modelling movement itself and instead model state machines that transition on events. Thus, both of these related projects are better suited for applications which involve reactivity and sensing of the environment, while Improv is better suited to applications where the user wishes to quickly generate certain movements and creatively explore movement patterns and transformations. We also have designed the Improv language with accessibility and ease-of-use in mind, especially for inexperienced programmers, and future work will focus on testing and measuring the usability of the system in user studies.

Architecture Overview

The *Improv* system consists of the following components:

- a live-coding interface, where the user writes a program in any text editor of their choosing. When they save changes to the file, they observe the resulting movement pattern on a simulated robot
- a shell script which monitors the user's file for changes, at which point it resets the simulator, and executes a compiled Haskell program which interprets the user code and starts a corresponding ROS node
- the ROS node, created with roshask [3], which sends messages to the simulator. Currently, only `Twist` messages can be created, which contain linear and rotational velocity information.
- A simulator (or actual robot), which recieves ROS messages and executes movement on a robot platform. Currently, we have tested the system with:
  - TurtleSim: a two dimensional simulator with limited physics, where velocity commands are nearly exactly executed on an animated turtle.
  - Gazebo with a TurtleBot robot model: a three-dimensional simulator with more realistic physics, where velocity commands control simulated motors.

See figure 1 for a diagrammatic representation of information flow in the system.

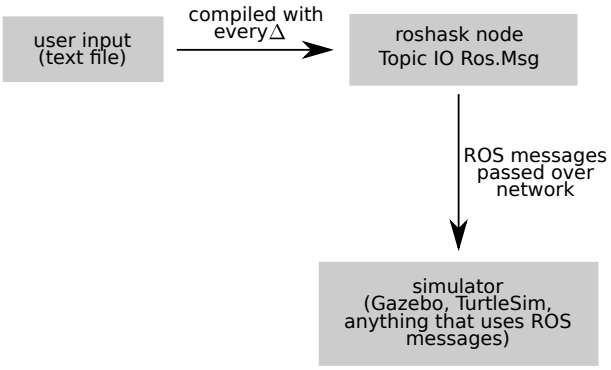

Figure 1: An illustration of how user input, written to a text file, is converted into a ROS node which publishes messages to a simulator.

Live-Coding Interface

One important design decision for developers of interactive text-based programming tools is whether to tie their tool to a specific editor. For example, the algorave live-coding tool TidalCycles was originally developed for emacs, a powerful editor which has a notoriously steep learning curve. Many people prefer to use simpler editors, so new live-coding plug-ins have been developed for editors such as Atom and Sublime Text. This editor-based approach has advantages, such as a large degree of customizability and extensibility using the features of the editors. However, it also introduces challenges such as maintaining feature parity between editors, as well as the up-front investment needed to interface with new editors. In our case, we are developing a tool that should be usable by artists, children, and programming novices, as well as experienced roboticists. Thus, we wish to allow users flexibility to use their editor of preference.

To accomplish this, instead of creating an interface for each desired editor, we use a shell script which monitors the file that the user is editing for

changes. Every time the user saves changes to the file, the program detects a change, interprets the user's new program, and resets the simulator and ROS node. This design choice circumvents the need to interface with specific editors. Additionally, many editors have keyboard shortcuts for saving files. Thus, executing programs contributes minimally to the overall workload of using the system, especially when compared to the many steps required to test changes to traditional ROS programs.

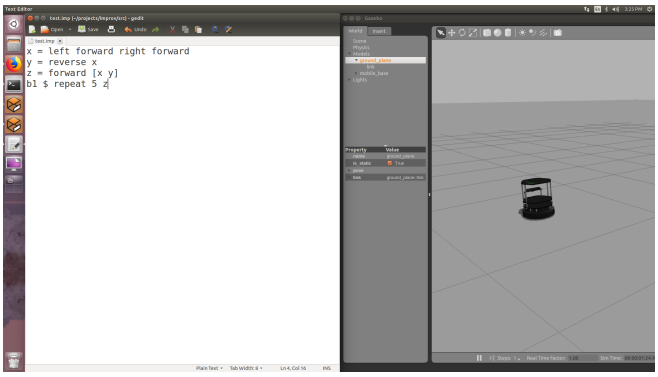

Figure 2: Examples of editor-simulator combinations possible in Improv. Note that Improv is entirely editor-agnostic (compiled when changes to file are saved). It is compatible with any ROS-compatible simulator and robot, though only message types for planar translation and rotation have been implemented so far.

Domain-Specific Language Design

The base type of the *Improv* language is a movement. Movements are discretized and can be combined with each other in various ways, forming different movements. The precise way in which this is interpreted on a robot platform is defined by the language's translation to Haskell and the commands sent to ROS for the particular robot in question.

The grammar of Improv.

The terminals in *Improv*, such as `forward` or `right`, are mapped to streams of ROS messages. In our implementation so far, we have mapped to the `Twist` ROS message, which has the type `Twist = {Vector3 linear, Vector3 angular}`, where `linear` and `angular` are three-dimensional vectors representing the robot's velocity. Here, we have simplified the language by varying only three of these values, the robots $x, y-$velocity in the plane and its angular velocity in the plane. Many ROS- and Gazebo-integrated robots have available velocity controllers. We have also discretized velocities, though users control velocity only relative to other movements, by specifying their relative timing.

Movements are organized in time into units, where each unit is performed in one "beat." The base timing of beats can be specified by the user, and is only limited by the maximum publishing frequency of ROS.

Users can specify a series of commands such as *move forward for one beat, turn right for one beat, move forward for one beat* with the command `forward right forward`. Movements separated by white space will occur in different "beats."

The user can also use brackets to compress a sequence of movements into one beat, such as `[forward right forward]`, which will cause these three movements to happen in the same amount of time as the first movement in the previous example. In our implementation, bracketing n movements causes each movement to be performed n times faster, but for $1/n$ times as long, so the movement has the same extent but is performed faster.

Movements can also be performed in parallel, such as `forward || right`, which will cause the robot to curve to the right as it moves forward. In our implementation, velocities in parallel are simply added, so `forward || backward` would result in no movement. A movement which is two movements in parallel will terminate when the "shorter" movement ends, so the program `(forward right) || forward` will never turn right (parenthesis group movements without changing timing).

We have also implemented several transformations which map a function over a movement. For example, `repeat` takes an integer and a movement as arguments and causes that movement to repeat for the specified number of times.

In space, we have `reflect`, which takes a plane and a movement as arguments and returns the reflected movement (`reflect YZ right` yields `left`, where the `YZ` plane is body-centered and would be the saggital plane on a biological organism). For transforming movements in time, `reverse` is a unary operator which will reverse the order of a series of movements: `reverse (forward right left)` is equivalent to `left right forward`. To reverse the trajectory itself, we use `retrograde`, which uses spatial reflections and reverses time; for example, `retrograde (forward right left)` is equivalent

to `right left backward`. Both `retrograde` and `reverse` are their own inverses: applying them twice returns the original movement, as would be expected.

While we have only implemented these combinators for simple and very symmetric mobile robots, one could imagine making more complicated types of symmetry for other robot platforms. We have included a typeclass `Symmetric a` which is defined by a function `refl :: Plane -> a -> a` which is parameterized by a body type `a`. By defining this function once for a new robot platform, detailing all the different symmetries of the body, the functionality of these spatial transformers can be extended to new platforms.

Multiple Robots

The *Improv* system has the capapbility to control multiple robots at once, using a syntax which mirrors how *TidalCycles* allows for multiple tracks to be composed simultaneously. Each robot is given a unique name in the shell script which launches ROS and the *Improv* system, and this is also where the initial location of each robot is specified. Then, in the user's program, they specify which movement sequence should be associated with each robot. For example, to make robot `r1` move forward and robot `r2` move backward, the user would write

```
r1 $ forward
r2 $ backward
```

This syntax, along with assigning movements to variables, can make it easy to specify relationships between how different robots are moving, such as

```
x = left right [forward right]
r1 $ x
r2 $ retrograde x
```

which would cause robot `r2` to perform the same movement as robot `r1`, but in retrograde.

Abstracting Movement in Haskell

Programs in the *Improv* DSL are interpreted by a compiled Haskell program into an ADT we call a `Dance`, which can be thought of as a tree that holds all movement primitives (the terminals in the *Improv* language, such as `forward`). The operators of the type encode the parallel/series structure of the user's program. To execute a `Dance` as a series of ROS messages, we must flatten the tree while maintaining this relative timing information, which will be discussed in Section SECTION.

`Dance`s are defined as

```
data Dance b = Prim Action Mult b
             | Rest Mult
             | Skip
             | Dance b :+: Dance b
             | Dance b :||: Dance b
```

where `Prim` is a motion primitive type, holding the `Action` (direction and spatial extent of the movement), `Mult` which stores timing information (to be described in Section SECTION), and `b`, a parameterized type describing the part of the robot to move. `Rest` indicates that the robot part is not moving for some period of time (and is a terminal in the *Improv* language). `Skip` is the identity dance, having no effect on the robot for no time duration, and is necessary for the monoidal structure of the parallel (`:||:`) and series (`:+:`) operators.

Algebraic Structure of `Dance`s

This algebraic structure helps enforce the timing behavior that we expect; namely, associativity. If `d1`, `d2`, and `d3` are all `Dance`s (with an arbitrary number and structure of movement primitives in each), then the order that they are sequenced together shouldn't matter:

```
(d1 :+: d2) :+: d3 = d1 :+: (d2 :+: d3)
```

And similarly, if they are all three in parallel, arbitrary groupings should not change the meaning of the program. This is exactly the behavior that the algebraic objects *monoids* have, namely associativity and an identity element. See [4] for a much more detailed discussion on the usefulness of monoids in modelling and programming languages, in the context of *Diagrams*, a Haskell DSL for creating vector graphics.

We can create monoid instances in Haskell for these operators on `Dance` data types, which allow for lists of `Dance`s (read in by the parser as all elements between square brackets or separated by the `||` operator) to be combined in sequence or parallel. The functions for doing so, `seqL` and `parL`, have the type `(Parts a) => [Dance a] -> Dance a`: they combine movements using the monoidal operator.

Relative Timing

As programs are parsed, we must enforce the timing semantics - movements inside square brackets, such as `[forward right forward]`, must occur within one "beat." To accomplish this, the parser reads in

the elements inside brackets as a list of `Dance`s.

Then we call a function `seqL` which uses the length of the initial list to determine how much to speed up each individual dance before composing the movements. This is accomplished with a function `changeTiming` which takes a multiplier `m` and a `Dance`, and propagates the multiplier through the `Dance` recursively. This allows for nested sequential movements: for example, the program `[forward [left left] forward]` would translate to the `Dance`

```
Prim (A f 3 r) :+: Prim (A l 6 r) :+:
Prim (A l 6 r) :+: Prim (A f 3 r)
```

where `f` is the `Action` corresponding to `forward`, `l` is the `Action` corresponding to `left`, and `r` is the robot body. Note that the two primitives inside the nested brackets have `Mult`s of 6, since they must occur six times as fast as normal to allow the whole movement to occur in one "beat."

```
data Direction = Lef | Righ | Forward
               | Backward | Center
               | Low | Mid | High
               | Direction :*: Direction

data Length = Zero | Eighth | Quarter
            | Half | ThreeQuarter | Full

data Action = A Direction Length
-- transforms actions (spatial)
transform :: (Parts a) => (Action -> Action) -> Dance a -> Dance a
transform f (x :+: y)      = (transform f x) :+: (transform f y)
transform f (x :||: y)     = (transform f x) :||: (transform f y)
transform f (Rest m)       = Rest m
transform f (Prim act m b) = Prim (f act) m b
transform f Skip           = Skip
```

- Propogating timing information down the tree

Interfacing with ROS

- how to extend to other platforms

Roshask is a client library for ROS, written in Haskell [3].

To integrate a robot with *Improv* for the first time, one must provide a function which converts `Dance` data types to ROS message types. This is largest barrier to extending *Improv* to other types of robots.

Example Programs

Conclusion and Future Work

- reactivity and sensing
- extending to other platforms: ECLs

With all of these implementation details, we would like to emphasize that the design decisions for how *Improv* programs are realized on robot platforms are relatively arbitrary and a single robot could have a multitude of different implementations. "It is not technological constraints that hold us back from using technology in new ways; technology changes at a tremendous rate. Our willingness to explore beyond the constraints of our imagination has the greatest effect" [5].

Acknowledgements

References

[1] L. Huang and P. Hudak, "Dance: A declarative language for the control of humanoid robots," Yale University, YALEU/DCS/RR-1253, August 2003.

[2] A. McLean and G. Wiggins, "Tidal–pattern language for the live coding of music," in *Proceedings of the 7th sound and music computing conference*, 2010.

[3] A. Cowley and C. J. Taylor, "Stream-oriented robotics programming: The design of roshask," in *Intelligent robots and systems (iros), 2011 ieee/rsj international conference on*, 2011, pp. 1048–1054.

[4] B. A. Yorgey, "Monoids: Theme and variations (functional pearl)," in *ACM sigplan notices*, 2012, vol. 47, pp. 105–116.

[5] T. Schiphorst, "A case study of Merce Cunningham's use of the lifeforms computer choreographic system in the making of trackers," Master's thesis, Simon Fraser University, 1986.