Improv - live-coding for robotic movement design

Alexandra Q. Nilles

February 12, 2018

Abstract

We introduce a new software tool, Improv, a programming language for high-level description of robot motion, integrated with immediate simulation of the resulting movement ("live-coding" for robots). The system is composed of a domain-specific language, which compiles to instructions which control the robot, either in simulation or reality. Simulation is done through a Haskell client for ROS. ROS ("Robot Operating System") is an open-source robot software framework which is widely used in academia and industry, and is supported by many commercially available robots. The compilation step is performed whenever the user saves changes to their program file, creating a "live-coding" interface which works with any text editor and provides immediate visual feedback in the robot simulator. The domain-specific language is inspired by choreographic techniques, and allows for several ways of composing and transforming movement primitives, such as reversing movements in space and time, and changing the relative timing of movements. Currently, *Improv* can be used to control any robot which uses the 'Twist' message type in ROS, and has been tested with TurtleBot robots in the Gazebo simulation engine. The intended users of this tool are anyone looking for an accessible way to generate robot motion, such as educators, artists, and researchers.

Introduction

- what is it
- possible audiences

Related Work

- dance notations
  - LBMS, lifeforms

There is a large body of work on formalizing and mechanizing notations and languages for describing and generating movement..

Over the last century, the Laban Bartenieff Movement System (LBMS) has been developed by a community of movement researchers. Labanotation has been developed as a movement notation system, used for precisely describing motion in time and space. The notation is specialized for human bodies, with symbols for specific body parts.

There is a notation for labelling a section of the score with a letter, and then later referencing that letter along with a repeat symbol to indicate which movement should be repeated. If there is no letter given, it is assumed that the most recent phrase is the one that should be repeated.

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.

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.

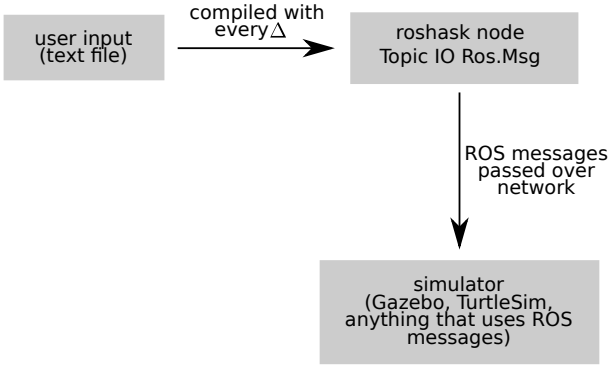

Figure 1: An illustration of how user input, written to a text file, is converted into a ROS node which publishes messages to a simulator.

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.

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 terminals in *Improv*, such as `forward` or `right`, are mapped to sequences 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`.

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 into one movement without changing timing).

We have also implemented several transformations which map a function over a movement. For example, we have `reverse`, which will

- combinators (repeat, reflect, reverse, retrograde)

Multiple Robots

The grammar of Improv.

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" [4].

Abstracting Movement in Haskell

- `Dance` and `Action` types

```
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

data Dance b = Prim Action Mult b
             | Rest Mult
             | Skip -- id for series, parallel
             | Dance b :+: Dance b -- in series
             | Dance b :||: Dance b -- in parallel
-- transforms actions (spatial)
transform :: (Parts a) => (Action -> Action) -> Dance
transform f (x :+: y)      = (transform f x) :+: (tra
transform f (x :||: y)     = (transform f x) :||: (tr
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

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] 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.