CKBot Platform for the ICRA 2010 Planetary Challenge

Shai Revzen, Jimmy Sastra, Nick Eckenstien, Mark Yim

I. Introduction

The ICRA Planetary Contingency Challenge 2010 will include 3 teams that will be using the new CKBot module and software package. This paper will present some of the new aspects of this hardware, underlying software architecture for quickly programming the modules and will include a tutorial for these teams.

II. Hardware

The planetary contingency will feature 50 CKBots and 10 CKBot wheel modules. The CKBots are manually reconfigurable modular robots that resemble a 6 centimeter cube on each side.

CKBots are aimed to be robust, using reliable connectors and high flex life ribbon cable. Cost is kept low by using hobby servos and making them easy to manufacture using lasercut ABS. They support a highly versatile range of locomotion as they can be configured into many shapes and have high torque capabilities. They are designed to be small enough to crawl through a 3 inch pipe, while having enough torque to cantilever 7 modules by itself. Capabilities can be improved by a spring loaded weight compensation mechanism that is easily attached such that it can create an even longer arm.

TABLE I
CKBot Specifications.

Size:	60x60x60mm.
Torque:	417 oz.-in.
Speed:	0.17s / 60 degrees
Communication:	CANBus, IrDa (local/remote range)
Weight:	146g.
Configuration:	manual with screws
Input Power:	24V

Each module has two micro controllers. One is tasked with CANbus and servo communication, the other is equipped with 7 independent serial ports and manages IR communication. IR communication features a local mode where communication only occurs between modules that are attached as well as a remote range mode where communication can occur at a distance of 2 meters if in line of sight. The local mode is useful for configuration recognition. The remote range mode can be used between unconnected clusters or modules connected through passive pieces.

Users can program CKBots using Robotics Bus, a protocol developed on top of CANbus. This software is written in Python.

A. Gravity Compensation

There is a new attachment to the ckBot module that can be used to compensate for gravity. It makes use of a spring attached to the active arm of the module to compensate for the weight of one or more modules. It should be noted that due to technical limitations, it can only be applied to one module in the vertical plane. However, it still makes certain tasks possible that were previously not possible. Most notably, use of an outwardly extended ckBot chain is aided by this attachment, allowing a longer arm before the motor at the base is not able to lift the chain. The attachment simply screws onto the side of the existing ckBot frame, and comes ready to attach. Figure **??** shows one module with a two module load and perfect passive gravity compensation over the full range of motion (180^o).

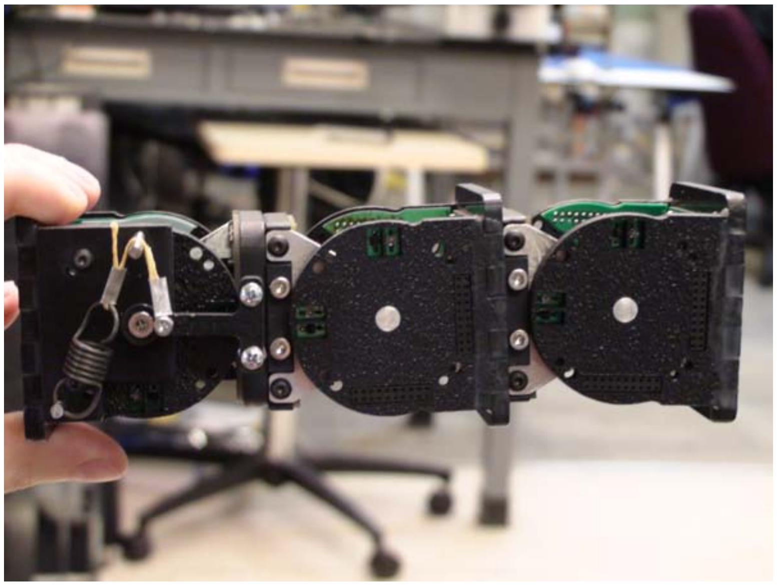


Fig. 1. The left most module has a passive spring gravity compensation attachment..

III. Software

The CKBot modules are controlled through software comprising several layers. At the lowest layer, modules support a Robotics Bus (RB) protocol layered on top of a CAN bus. RB defines means for nodes to be queried regarding their settable and gettable properties.

On top of RB we developed a python library that dynamically discovers and presents a logical view of a cluster of modules connected to a single CAN bus. The Cluster class can automatically discover properties of the modules and exposes them to the user via set and get methods in dynamically generated classes. When the module software version is familiar, the Cluster instantiates the appropriate Module subclass to represent it in addition to the automatically discovered properties. For interactive use and prototyping we designed the class structure to facilitate easy use from the Interactive Python shell which supports features like tab-completion for set and get methods and module names. For programming use, the discovered properties, low-level RB addresses and module sub-class methods are all presented with a combined naming scheme allowing them to be listed, read and written. The interface also provide fast communication with the modules using asynchronous IO. An example of the Cluster API is shown in Figure 2

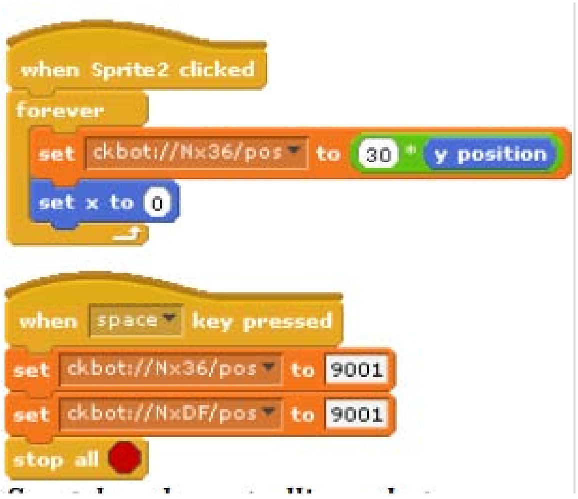
```
# Create a Cluster
c = Cluster()
# Expect 4 modules and name 3 of them as specified
c.populate(4,{ 0x91 : 'left', 0xb2 : 'head',0x5d :
'right'} )
# cluster.at provides logical alias for a numeric
module ID
assert c.at.head == c[0xb2]
# Set position of a head servo
c.at.head.set_pos(4500)
# cluster.od exposes auto-discovered properties
c[0xb2].od.set_pos(4500)
```

Fig. 2. Example of Cluster API use.

On top of the Cluster interface, we developed a GUI based on the wxPython cross-platform UI library. It presents a tree view of all properties in current modules, and allows the property lists to be quickly customized by editing them as YAML files. The user may select properties to examine and set via the UI, and receives visual indication if modules disconnect.

For high-level programming and interactive operations, we developed an application framework (JoyApp) based on pygame's event driven architecture and sporting an interface to the Scratch visual programming environment as shown in Figure 3. We enhanced pygame with new event types representing robot module position changes and Scratch events and sensor updates. JoyApp provides a powerful new abstraction: the Plan. A Plan is a sequentially executing behavior with its own incoming event queue and event handler. Plans may be executed sequentially or in parallel using a cooperative multitasking architecture that obviates the need to worry about thread safety.

A library of Plan classes is provided, including plans with the ability to load spreadsheets saved in the commonly supported CSV format as Gait Tables, Figure 4. A Gait Table is a spreadsheet whose columns are mapped by each SheetPlan instance to settable properties in the Cluster and whose rows represent consecutive times. A value in a cell represents the operation of setting the property to the specified value at the specified time. The use of a full-featured spreadsheet program



```
when Sprite2 clicked
forever
  set ckbot://Nx36/pos to 30 * y position
  set x to 0

when space key pressed
set ckbot://Nx36/pos to 9001
set ckbot://NxDF/pos to 9001
stop all
```

Fig. 3. Example of Cluster API use ..

to generate the gait tables makes it particularly easy to edit gaits and to explore gaits with a mathematical relationship between the values written in different times and into different properties.

"t"	"elbow"	"wrist"
0	1000	
0.5		1000
1	500	0
1.5	0	

Fig. 4. A "Gait table".

```
# … in JoyApp.onStart()
self.ltPinch=SheetPlan(self,
     loadCSV("gait.csv"),
     elbow="Nx35/pos", wrist="Nx55/pos" )
self.rtPinch=SheetPlan(self,
     loadCSV("gait.csv"),
     elbow="Nx5A/pos", wrist="NxA3/pos" )

# … in JoyApp.onEvent()
if evt.type==JOYBUTTONDOWN:
  if evt.button==0: self.ltPinch.start()
  elif evt.button==1: self.rtPinch.start()
```

Fig. 5. Using the same gait table on two different arms.