DARPA Programmable Matter Report Programming Chained Robotics in the Gas Programming Language

Makani Power

April 25, 2009

1 Introduction

Chained robotic systems are becoming increasingly prevalent, but programmability is a major barrier to their deployment. Present systems force programmers to think in terms of individual agents. Application code becomes entangled with details of coordination and robustness and often does not compose well or translate to other domains. We offer an alternate approach whereby the programmer controls a single virtual *spatial computer* which fills the environment space. The computations on this spatial computer are actually performed by a large number of locally-interacting individual agents. This abstracts the actual computational hardware behind the spatial computer interface, and allows the programmer to focus on a single model of global computation. We achieve this abstraction with two components: a language that embodies continuous space and time semantics and a runtime library that implements these semantics approximately. We demonstrate the efficacy of our approach with chained robotic algorithms on a running example of an inch worm.

We are investigating the programming of the motion of one dimensional linear chains occupying 3D space and forming three dimensional shapes. The chains are geometric polyhedral monomers connected by mechanical joints. We are currently focussing on two space packing parts: cubes and right-angled tetrahedra. We are using universal joints to connect neighboring cubes and hinge joints to connect neighboring tetrahedra. In this paper, we focus on the cubic geometry.

In order to make our chain smart, we assume that we have embedded compute and communication along with sensing and actuation in each monomer. We provide general serial processors capable of floating point processing and communication packets bounded but reasonably sized.

2 Programming Model

In this section, we introduce the Gas programming language, and build up facilities that support high-level chained robotic programming. In Gas, the computational model is based on manifolds of space that execute code, called the Amorphous Medium. The medium has computational state and physical extent, both of which evolve over time. We assume that the medium is populated by an infinite number of agents, and each agent can only communicate with neighbors within a fixed distance. Programs for continuous regions are then run approximately on a discrete set of agents. Each agent runs identical code but their execution diverges due to differing local state and environment and interactions with neighbors.

Gas programs are written as expressions over fields, where fields are mappings from manifolds to values. Expressions are executed repeatedly, producing streams of fields. Computations are specified as a network of instructions that manipulate streams of field values. Behaviors are produced from angle fields guiding actuation of devices sprinkled along the body of the chained robot.

Nearby devices share state – each device has some neighborhood in which it can access the state of other devices. Information propagates through the medium at a fixed rate, so neighbor state is time-lagged proportional to the distance travelled. Neighborhood communication occurs using a best-effort approach with each device maintaining a table of most recent time stamped shared data from its neighbors.

We cannot, of course, build an amorphous medium, but we can approximate it. Thinking about a continuous material will help us with both robustness and portability, since many differences between platforms and changes during execution can be subsumed into the single, simpler issue of approximation.

Why this dataflow model, rather than the familiar linear sequence of instructions on a serial machine? The fundamental reason is that information takes time to get from place to place within a spatial computer. This makes it impractical to synchronize execution across a computer of non-trivial expanse: the streaming dataflow model allows different parts of the computer to do different things, and synchronize weakly through the propoagation of data across the expanse.

The instruction set is mostly composed of simple pointwise operations like literals and arithmetic. These are of little interest, as they translate directly to equivalent operations on individual devices. A few critical instructions, however, satisfy the special needs of spatial computing. These are **restrict**, which limits the area in which computation is evaluated to effect branching, **delay** which time-shifts values, allowing state to be established in terms of feedback loops, and three groups of neighborhood operators: one selects sets of neighborhood values, another operates on those sets, and a third summarizes neighborhoods down into a single value. Figure 2 outlines the basics of Gas broken into ten categories. Gas is largely based on Proto and thus to learn more about Gas consult [3] for more information on the Proto programming language, [1] for a description of its implementation as a spatial computer, and [2] for Proto's usage in swarm robotics.

literals – *are self evaluating:*

`#t` → `#t`
`#f` → `#f`
`1` → `1`

tuples – *produce tuples of evaluated expressions and can be nested arbitrarily:*

`[1, 2]` → `[1, 2]`
`[[1, 2], 3]` → `[[1, 2], 3]`
`[1, 2][0]` → `1`

introspection – *provides point properties, where* `comm_range()` *produces the communication range,* `id()` *produces the point's unique number and* `dt()` *produces the time since last execution:*

`comm_range()` → num
`id()` → num
`dt()` → num

sensors – *provide sensory data:*

`button()` → num
`light()` → num

actuators – *drive the agent towards target states:*

`rgb([button(), 0, 0]` → *tup*
`servo(pi()` → `pi()`
`set_friction(100)` → 100

conditionals – *produce selected expression, but where* `?` *evaluates both branches and* `if` *evaluates only the taken branch:*

`#t ? [1 2] : [2 3]` → `[1 2]`
`if (#f) red(1) else blue(0)` → `0`

pointwise operators – *provide random, trigonometric, and arithmetic functions:*

`pi()` → π
`inf()` → ∞
`rnd(-1, 1)` → num
`-pi())` → $-\pi$
`sin(rnd(-pi(), pi()))` → num
`1 + 2` → `3`
`[1, 2] + [2, 3]` → `[3, 5]`

functions – *create functional abstractions, where* `fun` *produces anonymous and* `def` *produces named functions:*

`fun(x) { x * x } (2)` → `4`
`def sqr (x) { x * x }` → fun
`sqr(2)` → `4`

temporal integration – *maintains and combines values over time using feedback loops with initial and update expressions with actual updates occurring after computing all expressions:*

`var t = 0; t <- t + dt()` → num
`once(x)` → `var y = x; y`

spatial integration – *summarizes neighborhood values using* `@` *expressions, with* `hoodmin` *for minimizing,* `hoodmax` *for maximizing,* `hoodint` *for integrating over neighborhood expressions, and where* `lag@` *is the time since receipt of last data from neighbor, and* `vec@` *is the relative vector to the neighbor.*

`hoodmin(d@)` → num
`hoodmax(t@ + lag@)` → num
`hoodint(vec@)` → num

Figure 1: Overview of Gas Basics. All expressions evaluate to and operate on streaming fields.

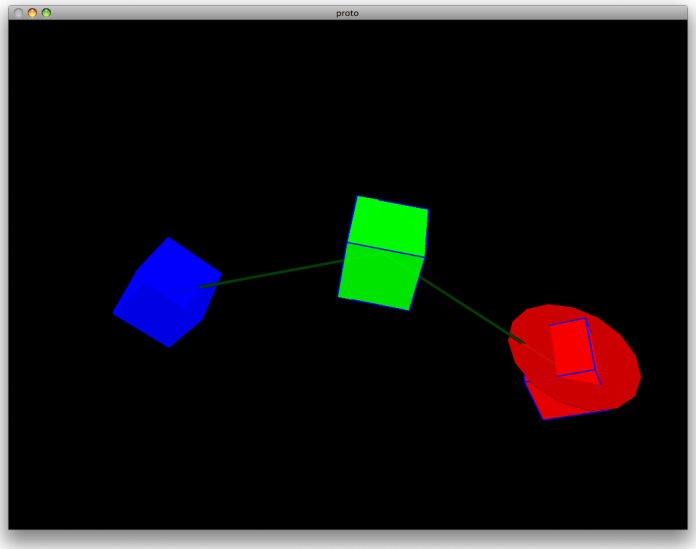

Figure 2: Three cube inchworm.

3 Inchworm

We introduce the inchworm as a running example that guides us through the use of Gas in writing chained robotic programs. The inchworm has an ingenious locomotion strategy. It is a worm with effectively two feet: a front foot and a rear foot. It moves by adjusting the friction on its feet while arching and relaxing its center segment which we refer to as its back.

3.1 Three Cube Inchworm

The simplest inchworm is comprised of three cubes with hinge joints between them as shown in Figure 2. Figure 3.1 shows the code and the following describes it in detail.

In order to control the inchworm we need to first construct a body map. We divide the inchworm into a rear foot segment, a back segment, and a front foot segment. We introduce the predicates `is_front` and `is_rear` that identify the front and rear feet with the back being any device that is not a front or rear foot. `is_front` takes as argument the number of devices in the chain. The following is the code for the body map:

```
def is_rear () {
 id() == 0
}

def is_front (n) {
 id() == n - 1
}
```

where `id` is the device id starting at `0` at the rear and ending at `2` at the front.

We can arch the back of the creature by servoing the hinge joints to positive angles and can relax the back by servoing to zero. We introduce the `servo` function that takes

```
let t = 4 * time();
servo(2 * (sin(t) > 0));
let segment = if (is_front(3)) -1 else if (is_rear()) 1 else 0;
let is_stopped = (segment * sin(t)) > 0;
set_friction(100 * red(is_stopped))
```

Figure 3: Simple three cube inchworm.

a target joint angle as its only argument. We can use a square wave to drive the arching with amplitude being the desired arch angle and frequency derived from a time signal as follows:

```
var t = 4 * time();
servo(2 * (sin(t) > 0))
```

where `time` is the synchronized time in seconds as described below, `t` is the square wave input, and where a sin wave is converted into a square wave by using the `>` predicate.

Every device has an internal free-running clock. We can synchronize time by having the device with the fastest clock drive the time of the entire ensemble of devices as follows:

```
def time () {
  var t = 0;
  t <- hoodmax(t@ + lag@)
}
```

where time starts out at zero and from there time computed as the maximum over the neighbor's time plus the staleness of the time (e.g., `lag@`).

We can adjust the friction of the inchworm's feet by using the `set_friction` function that takes a friction value as its only argument. We change the friction at the same speed as the arching by flipping between turning on or off the friction. We create the friction signal by creating a segment value of `-1` for front foot, `0` for back, and `1` for rear foot. This value can then modulate a sin wave and be clipped and amplified to create a friction signal as follows:

```
let segment = if (is_front(3)) -1 else if (is_rear()) 1 else 0;
let is_stopped = (segment * sin(t)) > 0;
set_friction(100 * red(is_stopped))
```

3.2 Many Cube Inchworm

A more general inchworm is constructed out of multiple cubes where the first and last cubes are the front and rear foot respectively as shown in Figure 4. In order to extend the three cube inchworm to a multiple cube body we need to generalize the body map and servo target angle.

In order to create a body map over an arbitrary length chain, we need to calculate the length of the chain. We can do that by first giving everyone a hop count from the rear foot of the worm. This count serves as a more general `id` and can be computed using a hop gradient as follows:

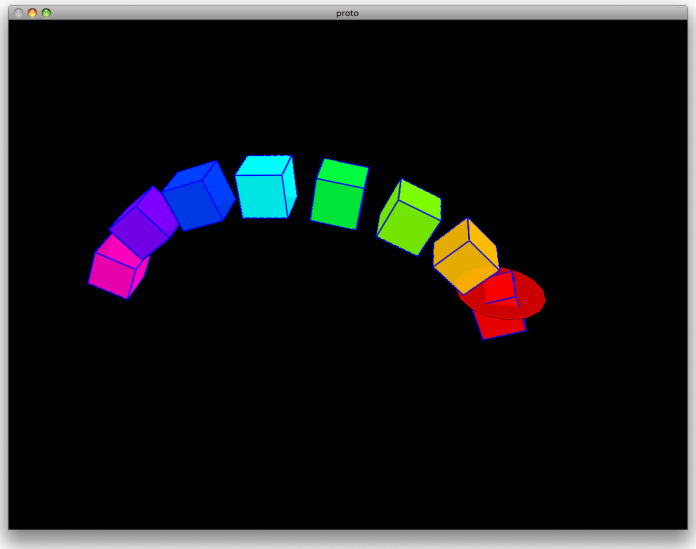

Figure 4: Many cube inchworm.

```
def hop_distance (src) {
  var n = inf();
  n <- src ? 0 : hoodmin(n@) + 1;
}
```

where the gradient starts at infinity and then is zero at the src (which is the rear foot) and otherwise is one plus the minimum over all neighbors' gradient values. This takes `N` time steps to settle for a `N` length chain.

Once we have hop distance from the rear foot, we can compute the length of the chain by broadcasting the highest distance value. We can find the highest distance value by finding the device with no neighbors with greater distance values as follows:

```
let k = hop_distance(is_rear());
hoodall(k@ <= k)
```

We can broadcast this value using a gradient based broadcast as follows:

```
gradcast(hoodall(k@ <= k), k)
```

where gradcast is as follows:

```
def gradcast (src, val) {
  var res = val;
  res <- src ? val : hoodmin_on(hop_distance(src), res)
}
```

where devices receive values from nearest sources by minimizing over `hop_distance`/`value` tuples using the `hop_distance` value as the key.

Figure 3.2 presents the code for the generalized many cube inchworm. The target servo angle and front foot are computed based on the number of elements in the chain.

```
let k = hop_distance(is_rear());
let n = gradcast(all@(k@ <= k), k) + 1;
let t = 4 * time();
servo((6 / n) * (sin(t) > 0));
let segment = if (is_front(n)) -1 else if (is_rear()) 1 else 0;
let is_stopped = (segment * sin(t)) > 0;
set_friction(100 * red(is_stopped))
```

Figure 5: Generalized many cube inchworm with length computation.

3.3 Robust Inchworm

In order to provide truly robust inchworms we would want to have redundant actuators and joints. Perhaps the actuators and joints would be separate mechanical systems with their own redundancy. Actuator devices would be sprinkled along a skeletal system and would communicate with multiple neighbors. Actuation would be based on local radius of curvature. The inchworm would arch using a target radius of curvature along its length so that the arch would form a perfect semicircle at its apex. Failed actuator devices would be naturally compensated by not including them in the local radius of curvature calculation.

4 Future Work

This work is just the beginning towards working on chained robotic programming. We will develop more programs for more scenarios and bring over concepts developed for Gas in other domains.

We will develop a general body language for describing chained robotic bodies. We would like to allow it to handle many joint types and branching structures.

Robustness has just been introduced in this report but is a very important topic that needs to be addressed more fully for all of our programmable matter research. In particular, we will examine how to make robotic chains built using redundant actuators along their body. We will also look at other types of failures and fault handling.

Finally, we will relate Gas to RALA and see if Gas might run on top of RALA in some workable fashion.

References

[1] Jonathan Bachrach and Jacob Beal. Building spatial computers. Technical Report MIT-CSAIL-TR-2007-017, MIT CSAIL, 2007.

[2] Jonathan Bachrach, James McLurkin, and Anthony Grue. Protoswarm: A language for programming multi-robot systems using the amorphous medium abstraction. *AAMAS 2008*, April 2008.

[3] Jacob Beal and Jonathan Bachrach. Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intelligent Systems*, pages 10–19, March/April 2006.