ESPRESSOCAD

A System to Support the Design of Dynamic Structure Configurations

MICHAEL PHILETUS WELLER[1], ELLEN YI-LUEN DO[1] AND MARK D GROSS[1]
[1]Design Machine Group, University of Washington

Abstract. EspressoCAD is a computer assisted design system created to support the design of structures composed of our modular robotic building blocks, Espresso Blocks. EspressoCAD's interface allows designers to manipulate individual blocks. By recording these manipulations designers can create actions to control the behavior of groups of blocks. The current version is implemented as an AutoCAD plugin written in Visual Basic.

1. Introduction

The class of modular robots known as 'crystalline modules' (Rus and Vona 2001) promises to provide the building blocks for a new design medium. A group of these robotic blocks can take any arbitrary shape to a resolution limited by the size of a one-block voxel, and then autonomously transition to another arbitrary shape (Rus and Vona 2001). Applying this technology to building blocks will allow designers to create dynamic objects that shift form through time. But specifying these dynamic forms will require new design tools.

We first attempted to use crystalline modules as building blocks with our Espresso Blocks (Weller 2003) architectural building system. EspressoCAD is a computer assisted design system we built to support the design of configurations for structures composed of Espresso Blocks. Dynamic structures composed of these blocks could be used, for example, to create urban live/work spaces that would adopt different configurations throughout the day. A live/work espresso stand built with Espresso Blocks could function as a coffee shop during the day and as an apartment at night. Espresso Blocks could also be airlifted to support quickly deployable shelters in hostile and distant places, or be launched into orbit to provide a reconfigurable structure for satellites and space stations.

Designing these dynamic structures will require CAD systems with new capabilities. Such a system must at least:

1) support the design and visualization of a structure's transition from one form to another
2) help designers manage the new degrees of freedom allowed by a dynamic structure by modeling the constraints of the structural system
3) output a file that can be loaded into a structure and used to transform it into the designed form

In this paper we present EspressoCAD, our first attempt to build a system that supports this minimal specification in order to demonstrate the potential power of dynamic structures. As EspressoCAD is designed specifically to support the design of structures composed of Espresso Blocks, we will begin with a brief description of the Espresso Block module in section two. In the third section we will explain how a designer can use EspressoCAD to design configurations for a group of blocks. We give the details of EspressoCAD's implementation in the fourth section. In the fifth section we outline our plans for our next generation of building blocks, Cocoa Blocks, and our system to support the design of local rulesets for Cocoa Blocks, CocoaCAD.

2. Espresso Blocks

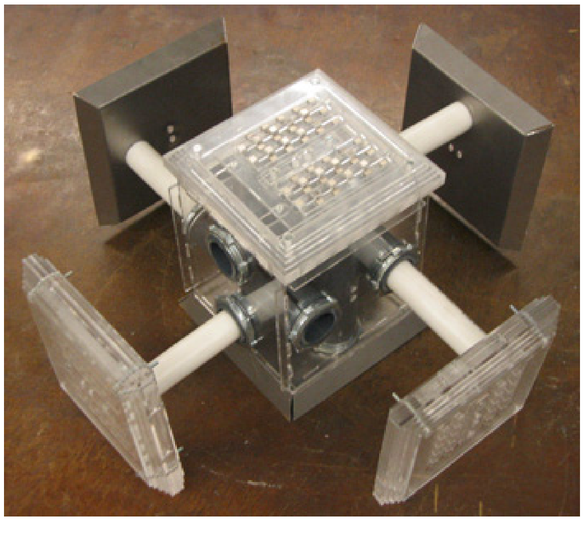


Figure 1: *Prototype Espresso Block module*

Our Espresso Block module (Figure 1) is a crystalline modular robot similar to the systems of Rus and Vona (2001) and PARC (Suh, Homans and Yim

2001), but on an architectural scale. While other modular robotics systems have focused on creating very small modules to support autonomous locomotive agents (Rus et al. 2002), Espresso Blocks are objects on the scale of a brick or concrete masonry unit, and are designed to stack themselves to create load-bearing structures (Weller 2003). Our current prototype block is 15cm wide with the faces contracted and 30cm wide with the faces expanded (Figure 2).

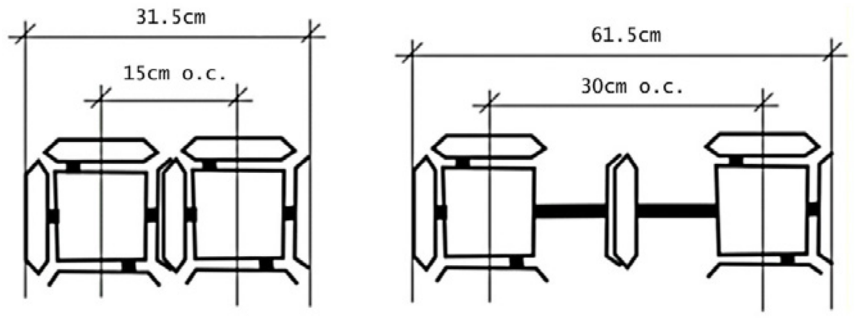


Figure 2: *Block module dimensions*

Figure 3 shows a section diagram of a block module. Three faces of each block are male, with a switching magnet array latch. The opposite three faces are female, with sheet metal dishes to receive the magnet arrays. Each face is mounted on an arm that fits into a tubular housing in the core of the block. We are currently experimenting with a rack and pinion drive to extend and retract the arms. When the arms of two adjacent blocks are extended towards each other, the switching magnet array from one block fits into the metal dish of the next block, latching them together. To unlatch from each other, the switching magnet array is switched to its 'off' configuration directing its magnetic field inwards away from the metal dish and both faces are retracted (Weller 2003).

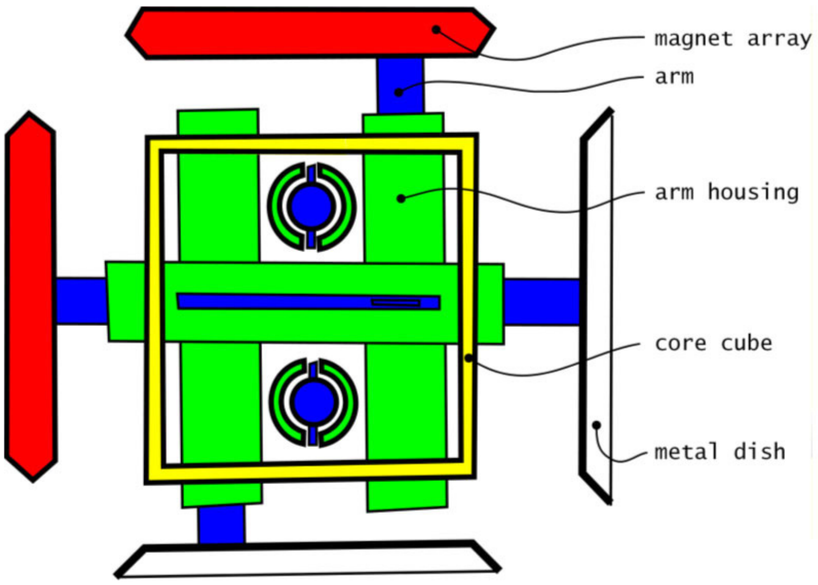


Figure 3: *Block module section diagram*

Pairs of Espresso Blocks move with an inchworm-like motion (Figure 4). One block, the free block, unlatches itself from all blocks that would constrict its movement, while the other block, the anchor block, remains latched to the rest of the structure. Both blocks extend their linked faces, pushing the free block across a one-block wide space where it latches to an adjacent block. To continue moving the anchor block can disengage from neighboring constrictive blocks to become the free block and both blocks retract their faces to bring the two blocks back together, shifted by one space. As Rus and Vona (2001) show, by repeating these steps a group of blocks can take any arbitrary form.

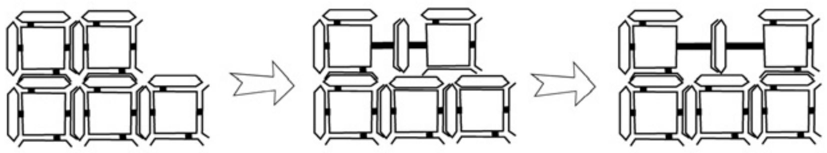


Figure 4: *Block inchworm motion*

Because a building composed of modular robots would be able to autonomously reconfigure itself into a new shape, it can be designed as a series of dynamically linked forms and behaviors rather than a single static

3D shape. One example of such a dynamic form is the live/work espresso stand (Weller 2003). This structure would erect itself into a one-room space from several pallets of blocks delivered to the building site. With a remote control (Figure 5), the stand's occupant could cycle through different forms, from an espresso stand in the morning to a kitchen and dining room in the afternoon, and then into a bedroom at night. To transition from one space to another the blocks would not only reconfigure architectural elements such as walls, counters and windows, but would also form furnishings for the room and switch items such as the espresso machine and alarm clock in and out of storage as appropriate.

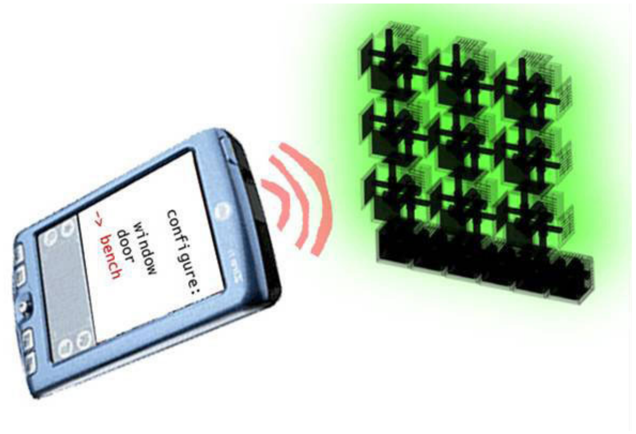


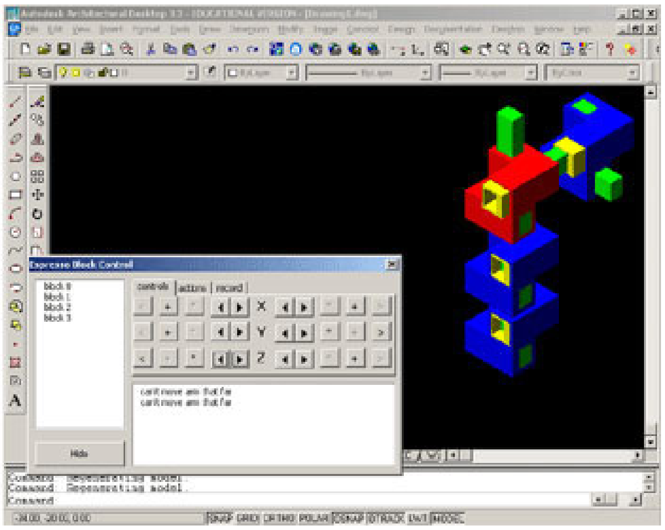
Figure 5: *Using a palm pilot as a remote control*

The live/work espresso stand's occupant would control the transition between forms by selecting and playing actions designed in EspressoCAD. With a remote control that could be a palm pilot or cell phone with the block control software loaded onto it, the occupant would select which area of the structure to reconfigure. After the occupant selects the desired action from a list, the blocks that will be involved in the transition light up green. Once the selection is confirmed, the blocks turn red and begin moving. When each block reaches its final position its light turns off to signal when the transition has completed.

3. Designing with EspressoCAD

Unlike other CAD programs that create drawings to guide contractors or 3D data to send to a rapid prototyper, EspressoCAD produces actions to be executed by groups of Espresso Blocks. The design interface (Figure 6) features a 3D view of the group of block modules and a control panel.

EspressoCAD represents block modules with a more abstracted form, a cube with male and female arms extending from opposite faces. Actions can be created from scratch by adding new blocks to the design space and then recording their manipulation, or existing actions can be opened, played back and extended. Completed actions can be saved and loaded onto a remote control, or uploaded onto a website to be shared with others.

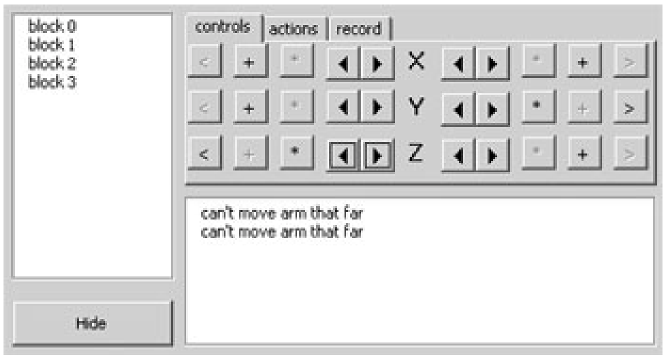


```
can't move arm that far
can't move arm that far
```

```
Command: Regenerating model.
Command: Regenerating model.
Command:
```

Figure 6: *EspressoCAD screen shot*

To help designers manage the additional degree of freedom allowed by change through time EspressoCAD models the constraints of the block modules. The control panel (Figure 7) has three tabs: the controls tab, the actions tab and the record tab. The controls tab presents the designer with a set of manipulations that can be played on the currently selected block and allows selection to be passed to an adjacent block. When a block is selected, it is highlighted on the screen in red, and control functions that are unavailable to that block are grayed out.



```
can't move arm that far
can't move arm that far
```

Figure 7: *EspressoCAD control panel*

There are an identical set of buttons for each axis of the selected block. The buttons on the left side control the male (green) face and the buttons on the right control the female (yellow) face. The arrow buttons immediately adjacent to the axis label extend and retract that face of the block. When the face of one block is extended to the face of an adjacent block, they automatically latch together. When the face of the selected block is latched to another block, the '*' button, which unlatches the face from the adjacent block, becomes active. If the space adjacent to the face is unoccupied, the '+' button is active and allows a new block to be added to the design space latched to that face of the selected block. If there is another block adjacent to the face the '<' button is active and allows selection to be passed to the adjacent block.

While EspressoCAD does not impose physical constraints such as gravity, it imposes the block freedom constraint to make it possible to show that the blocks' limited range of motion does not preclude the transformation between two configurations. Figure 8 shows the selected block marked with an 'x' attached to another block that is circled. If the selected block extends or retracts the arm connected to the circled block, and the circled block is free to move, it moves along with the arm. If the circled block attached to the arm is not free to move, but the selected block is, then the selected block moves and the circled block stays in place.

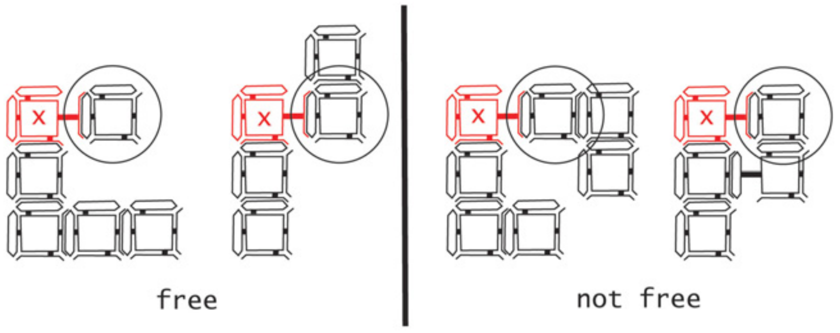


Figure 8: *Block freedom conditions*

The circled block is free to move if it is not attached to any blocks besides the selected block, or is only attached to one other block besides the selected block, and that other block is only attached to the circled block. If an arm of the selected block is extended or retracted, and both the selected block and the block the arm is attached to are not free, nothing moves, and an error message is printed to the message window. If an attempt is made to extend or retract an arm past its maximum or minimum bounds, the arm does not move and "can't move arm that far" is printed to the error window.

The record tab allows a new action to be created. Once a name for the action is entered, manipulations of the blocks are recorded to the action. After an action is recorded and saved, it can be placed on the actions tab. To add actions to a remote control, the actions tab would be downloaded onto the remote control device. By arranging a set of actions on the actions tab and playing them back in EspressoCAD, a designer can simulate the control of a block structure before attempting to play the actions on a group of blocks.

To show that a block module's limited range of motion is sufficient to allow a block structure to reconfigure itself, we designed the pallet-to-wall action (Figure 9). This action transitions a pile of blocks on a pallet into a two-block thick wall. The action only transitions one layer of the pile at a time, so by repeating the action four times an entire pallet could be transitioned. Starting with the far right column of a 4 x 4 pile, the blocks inchworm up and over to the left, until they form a 2 x 8 tower. This is a relatively simple action, but future dynamic structure configuration design systems will feature tools to manage more complex transitions.

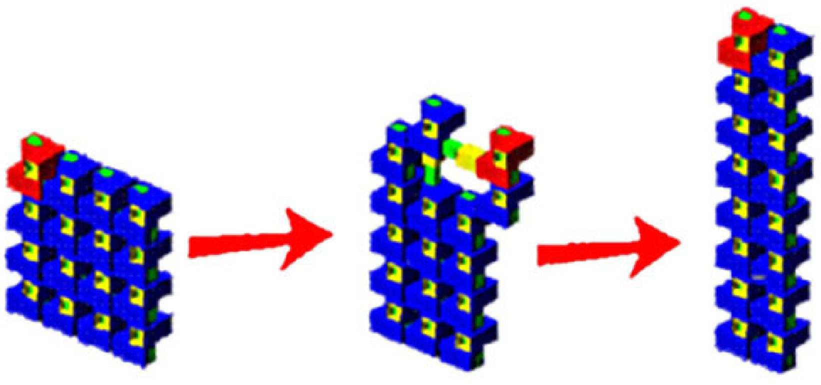


Figure 9: *Pallet-to-wall action*

4. EspressoCAD Implementation

EspressoCAD is built as an AutoCAD plugin to take advantage of its rendering engine and viewing controls. We initially explored writing the plugin in AutoLISP, but decided to use Visual Basic instead as the application protocol interface for 3D objects appeared much more straightforward. EspressoCAD is implemented as a series of objects. The world object holds a list of block objects, an associative array of the positions of all of the blocks, and a control panel object. Each block object contains six arm objects, one for each face. Pressing a button on the control panel routes the command through the world object to the current block object.

EspressoCAD manages locational relationships between blocks such as adjacency, latching and collisions by maintaining a multi-dimensional associative array of the positions of every block (Figure 10). The top level of the associative array contains an index of all of the x positions currently occupied by a block. Each x position contains a list of all the y positions occupied by blocks at that x coordinate. Each y position has a list of z positions occupied by blocks at those x, y coordinates. And each z position has a reference to the block object that occupies that position.

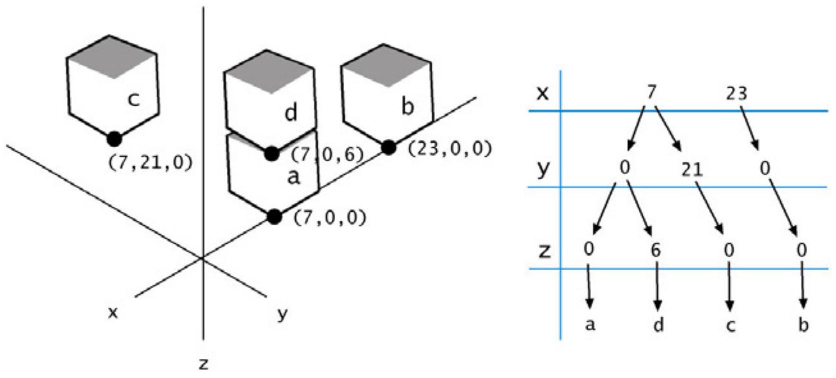


Figure 10: *Block position associative array structure*

The psuedo-code to test whether there is a block within a bounding box is shown in Figure 11. The 'blocks_within_bounding_box' function takes the minimum and maximum points of the bounding box as arguments. First it looks whether there are any position keys on the x position list that fall within the bounding box's x coordinates. If there are not, the test returns an empty list. If there are position entries within the bounding box's x coordinates, each of those entries' y position lists are tested for objects within the bounding box's y coordinates. Again, if there are no entries, the test returns an empty list. If there are entries, each of the entries' z position lists are tested for entries within the bounding box's z coordinates. If there are any entries found, the test returns a list of references to the blocks found at each entry. Each entry should have just one block because blocks are not allowed to overlap.

```
define blocks_within_bounding_box( min, max ):
  block_list = [] // list of blocks to return
  // check x position list
  for( x = min.x, x <= max.x, x++ ):
    if( position_array.x_list.has_key(x) ):
      // check y position list
      y_list = position_array.x_list{x}
      for( y = min.y, y <= max.y, y++ ):
        if( y_list.has_key(y) ):
        // check z position list
        z_list = y_list{y}
        for( z = min.z, z <= max.z, z++ ):
          if( z_list.has_key(z) ):
            block_list.append( z_list{z} )
  return block_list
```

Figure 11: *Psuedo-code to test for a block within a bounding box*

Every time an unlatched arm of a block is extended it checks to see whether it has latched to another block. To find out it asks the world object if there is another block within one block length. If there is a block within one block length, the world object returns a reference to it. The arm being extended then asks the adjacent block for its position and the extension of the arm facing it. If the combined extension of the two arms bridges the gap between the blocks they are latched together. Otherwise the arm is simply extended.

If the arm being extended is latched to another block the selected block calls the block to which the arm is latched to determine if it is free (Figure 8). The freedom constraint is an approximation of the physical constraints of an actual block structure. The psuedo-code for determining whether a block is free is shown in figure 12:

```
define is_free( block ):
   latched_list = [] // blocks latched to this block
   // check which blocks arms are latched to
   for each arm in block.arm_list:
      if( arm.is_latched() ):
         // add blocks besides selected block to list
         if( arm.latched_to_block() != selected_block ):
         latched_list.append( arm.latched_to_block() )
   // check the length of the latched_to list
   if( latched_list.length() == 0 ):
      // block is only latched to selected block
      return true
   else if( latched_list.length() > 1 ):
      // block is latched to at least two other blocks
      return false
   else:
      // block is latched to one other block
      for each arm in latched_list[0].arm_list:
         if( arm.is_latched() ):
            if( arm.latched_to_block() != block ):
               // other block is latched to more than one
               // block
               return false
      // other block is not latched to more than one
      // block
      return true
```

Figure 12: *Psuedo-code to test whether a block is free*

The 'is_free' function takes a block as an argument. It checks each arm of the block passed to it to see if it is latched to a block besides the selected block. If it is the block to which the arm is latched is added to a list of blocks, the 'latched_list'. If the block given as the argument is not latched to any blocks besides the selected block, it is free and returns true. If it is latched to more than one block besides the selected block, it is not free and returns false. If it is latched to only one other block it checks that block to see if it is attached to any other blocks. If it is it returns false, otherwise it returns true.

If the block is free it calls the world object to see if the space that it is going to move into is occupied by any other blocks. If the space is free the world object updates the block's position in the position array and tells both the arm and the block to which it is latched to move. If there is another block in the way, nothing moves and an error message is printed to the control panel's message window.

When the record button is pressed, it creates a function definition in a new file and tells the world object to record actions. Every time a button is pressed on the control panel the function call is written to the file. To add the function to the actions tab, a new button is added to the tab and linked to the new function.

5. Future Work: CocoaCAD

EspressoCAD demonstrates several features to support the design of dynamic objects, including modeling the design constraints of a block module and supporting the creation of actions. But managing complex inhabitable spaces will require design tools that can operate at a higher level than pushing individual blocks around. To take advantage of the distributed parallel computing power of the blocks and speed up the reconfiguration process, instead of manually planning each block’s path beforehand it would be more effective to broadcast the desired configuration or behavior to the blocks and let them work out the transition. A strategy to achieve this effect by using local rulesets to control block behavior is proposed by Jones and Mataric (2003). A design program takes a desired goal shape and generates local rulesets to be loaded onto the individual blocks. In a simulation, each block evaluates its local conditions and takes the actions required to satisfy its ruleset, more or less producing the desired goal shape without inter-block coordination. Shen, Salemi and Will (2002) switch their chain modules between locomotive gaits by sending “hormone” messages to cycle the individual modules between rulesets. For example, a system composed of a chain of several modules arranged in a loop would have all of its modules using the ‘roll’ ruleset, producing a rolling behavior in the entire system. By broadcasting a message telling all of the modules to switch to the ‘crawl’ ruleset, the system transitions into a snake-like shape and commences crawling forwards.

5.1. COCOA BLOCKS

We are currently working on a children’s building block kit to support ruleset design education, Cocoa Blocks. This kit will have three parts, a 3D modeling and simulation application to support ruleset design, a remote control to broadcast ruleset changes to the blocks, and the blocks themselves. The small pile of six inch blocks that comes with the kit will be suitable for designing a piece of furniture, or experimenting with different gaits to move around and over obstacles.

5.2. DESIGNING WITH COCOACAD

Instead of designing actions that explicitly list every step every block should take to transition to a goal shape, CocoaCAD will support the design of local rulesets that lead to the emergence of a desired goal shape from any arbitrary starting shape. In CocoaCAD goal shapes will be designed on an interface similar to EspressoCAD's, but instead of recording the manipulations as an action it will generate rulesets to create the shape. CocoaCAD's ruleset editing interface will then visually show the effect of manipulating and editing the generated ruleset.

A set of rulesets output by CocoaCAD would then be broadcast to the Cocoa Blocks. The local rules will tell the block to evaluate the position of immediately adjacent blocks and obstacles, and specify an action for the block to take if the local conditions match a rule. All blocks would have identical lists of local rulesets, but would select one ruleset from the list depending on the state of the block. A rule can tell a block to switch states to a different ruleset in the list. Rulesets could be designed to converge on one goal shape and then wait for a command to switch to a new form, or they could be used to create a dynamic form that would respond automatically to changes in the environment.

5.3. COCOACAD IMPLEMENTATION

We are building CocoaCAD as a freestanding python application to allow the more intensive processing that will be required to generate rulesets and simulate their effect on block behavior. The rulesets are being written in XML so that they will be able to be downloaded onto blocks and run, posted on websites, or sent over email.

We hope that deploying these block kits in an educational setting will produce valuable feedback on the ruleset design interface, to allow us to evaluate which features are most useful and to identify design obstacles that still need to be addressed. At the same time we hope Cocoa Blocks and CocoaCAD will begin to cultivate designers who are comfortable creating in this new design medium.

References

Butler Z, K Kotay, D Rus and K Tomita: 2001, Cellular Automata for Decentralized Control of Self-Reconfigurable Robots *in Proceedings of the IEEE Int. Conf. on Robotics and Automation (ICRA 2001), workshop on Modular Self-Reconfigurable Robots*, Seoul, Korea.

Jones C and M Mataric: 2003, From Local to Global Behavior in Intelligent Self-Assembly in Proceedings of the IEEE International conference on Robotics and Automation (ICRA 2003), Taipei, Taiwan. 721-726.

Rus D and M Vona: 2001, Crystalline Robots: Self-reconfiguration with Compressible Unit Modules *in Autonomous Robots (10)1*, January. 107-124.

Rus D, Z Butler, K Kotay and M Vona: 2002, Self-reconfiguring Robots *in Communications of the ACM 45(3)*, March. 39-45.

Shen W, B Salemi and P Will: 2002, Hormone-Inspired Adaptive Communication and Distributed Control for CONRO Self-Reconfigurable Robots *in IEEE Transactions on Robotics and Automation 18(5),* October. 700-712.

Suh J, S Homans, M Yim: 2001, Design Tradeoffs for Modular Self-Reconfigurable Robots: The Mechanical Design of Telecubes (A Case Study in Progress) *in Proceedings of the IEEE Intl. Conf. on Robotics and Automation (ICRA 2001) Workshop on Self-reconfigurable Robots*, Seoul, Korea.

Weller M: 2003, *Espresso Blocks: Self-Configuring Building Blocks*, Master of Architecture Thesis, University of Washington, Seattle.