

# **A Programmable Constructor in a Kinematic Environment**

William M. Stevens

Dept. of Physics and Astronomy, Open University, Milton Keynes, England, MK7 6AA

[william@stevens93.fsnet.co.uk](mailto:william@stevens93.fsnet.co.uk)

[www.srm.org.uk](http://www.srm.org.uk)

## **Abstract**

It has been conjectured that self-replicating systems will enable complex macroscale objects to be made in a reasonable time using nanoscale assembly processes [1]. Self-replicating systems might also be used within reconfigurable structures on any scale, such as micro-circuits.

An abstract model of a programmable constructor in a kinematic simulation environment is presented. This programmable constructor can form the basis of a self-replicating system in the same environment.

The simulation environment is a 3D moveable finite automaton in which cubic parts can be placed into a discrete space and connected together to make machines.

There are 6 types of part, each type performs one of the following simple logical and mechanical functions: signal propagation, NOR-function, rotation, connection, disconnection, motion.

The programmable constructor can take a disorganised collection of parts as its input and construct machines from these parts according to a sequence of instructions.

## **1. Introduction**

In a lecture given at the University of Illinois in 1949 [2], mathematician John von Neumann proposed a framework for studying the logical, kinematical and mechanical aspects of machines capable of constructing other machines. Later, he decided to concentrate on the purely logical aspects of the design of a universal programmable constructor, able to construct arbitrary automata within its environment of operation. Under the control of a particular program, this automaton could construct a duplicate of itself. The machine was an abstract mathematical entity, designed to operate within a cellular automaton environment. The design contained approximately 200,000 cells [3].

Several other researchers since von Neumann have investigated self-replication using cellular automata. E.F. Codd [4] produced a design similar to von Neumann's in a cellular environment with fewer states per cell. Starting from Codd's environment, Langton showed that self-replication becomes trivial if the requirements for universal construction and universal computation are dropped [5]. Langton's self-replicating loop consisted of 86 cells with 8 states per cell.

It must be mentioned here that different researchers have had different reasons for investigating self-replicating systems even though they have used similar methods and similar environments. As part of a larger endeavour to elucidate aspects of a general theory of automata, von Neumann was interested in designing an automaton able to construct *arbitrary* automata within a domain of operation. Small additions to such an automata enable it to construct a replica of itself. Langton was interested in demonstrating a very simple system having the capacity for self-replication and nothing more. McMullin [6] explains some of the differences between the goals of von Neumann and Langton.

There have been several studies of physical self-replication, and a handful of self-replicating devices have been built. Penrose's wooden blocks [7], and the electro-magnetic catalyst of Morowitz [8] are template-based replication schemes. In these systems, a particular configuration of two or more parts serves as a base upon which a replica is formed when the configuration is placed in a container full of parts and agitated. Chirikjian et al. [9] devised a robot capable of building a replica of itself from a small number of pre-assembled subsystems. In the three systems just described the input parts upon which the self-replicating devices operate are not much less complex than the devices themselves. Also, the devices are not designed to have any constructional capability. In these devices the problem of physical self-replication is a self-assembly problem in which the parent device catalyses the self-assembly of its offspring. This kind of self-replication is distinct from the kind of self-replication that arises as a special case of automated construction.

Other studies have addressed automated physical construction, often with self-replication as an ancillary goal. Moses [10] designed a set of 11 different types of block from which a constructing device could be built. This device operated on the same types of block from which it was built, and was controlled by a computer outside the constructor's domain of operation. Each block contained a single functional component. For example, there was a block containing a motor, a block containing a cog and a block containing a single-axis movable rod. Toth-Fejel [11], Murata [12], Zykov et al. [13] and others have proposed or built systems consisting of identical modules, each containing a microprocessor to control both communication between modules and movement of one module relative to another. In such systems, self-replication is a matter of providing a program  $P$  for the modules of a machine  $M$  which causes  $M$  to move in such a way as to cause an unorganised collection of modules to take on the same configuration as the machine  $M$ , and then copy the program  $P$  into the duplicate machine.

Earlier papers by the author [14,15] described two simulation environments called Nodes and CBlocks that can be regarded as being part way between physically unrealistic but easy-to-reason-about cellular automaton environments and physical environments with movable parts. CBlocks is a two-dimensional discrete space environment similar to the environment described in this paper. Nodes is a two-dimensional continuous space

environment. Both CBlocks and Nodes contain a part that is able to create any other part from nothing when fed with a signal encoding the part to be created. Self-replicating systems were demonstrated in both of these environments.

Freitas and Merkle's book 'Kinematic Self-Replicating Machines' [16] contains a comprehensive summary of all published work on self-replicating systems to date.

This paper presents an environment called CBlocks3D. In common with cellular automata environments, CBlocks3D allows the logical and geometrical structure of automata to be studied. CBlocks3D also allows some of the kinematical aspects of automata to be studied. This paper shows how a programmable constructor can be implemented in CBlocks3D, and how this constructor could be modified to implement a self-replicating system.

## **2. The CBlocks3D simulation environment**

CBlocks3D is a three-dimensional discrete-space, discrete-time moveable finite automaton environment. Cubic parts in this environment can be connected together to make machines. There are 6 different part types. Binary signals can pass between the faces of neighbouring parts. It takes one time step for a part to respond to an input signal. In a single time step, a part may move one unit in any one of six directions. When a part moves, all parts directly or indirectly connected to it also move. Systems with similar laws of motion have also been used by Arbib [17] and by Thompson and Goel [18].

Each part has the following properties:

- Type: one of the 6 types described in Table 1.
- Orientation : A part may be in one of 24 possible orientations. For parts which have rotational symmetry, some of these orientations will be indistinguishable.
- Connectivity : Each face of a part may be connected to the face of a neighbouring part.
- Outputs : The 'Delta' and 'Nor' part types can produce output signals in response to particular input signals.

Table 1 describes the function of each of the six types of part. In Table 1, the letters N,E,S,W,F and B are used to refer to the value of the input signal at the North, East, South, West, Front or Back faces of the part. North is up the page, South is down the page, East is to the right of the page, West is to the left of the page, Back is into the page, Front is out of the page.

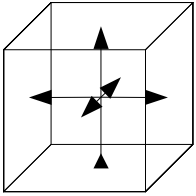
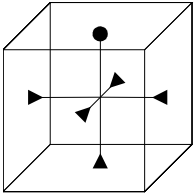
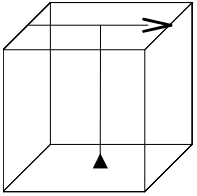
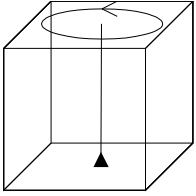
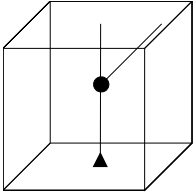
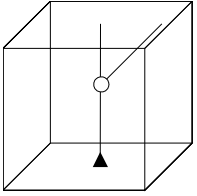
		
<b>Delta</b> N,E,W,F,B := S	<b>Nor</b> $N := \neg(E \vee W \vee F \vee B \vee S)$	<b>Slide</b> If S, slide part lying North one unit East
		
<b>Rotate</b> If S, rotate part lying North about the South-North axis in an anticlockwise direction.	<b>Fuse</b> If S, fuse parts lying North and North-East	<b>UnFuse</b> If S, unfuse parts lying North and North-East

Table 1. Part types in the CBlocks3D environment.

The Delta, Nor and Rotate parts have rotational symmetry about the North-South axis and can therefore be in any one of 6 behaviourally distinct orientations. The Slide, Fuse and Unfuse parts have no rotational symmetry, and can therefore be in any one of 24 behaviourally distinct orientations. The term 'main axis' is used to refer to the axis of a part on which the input lies, going from South to North in Table 1.

Table 2 shows two simple machines that can be made with these parts.

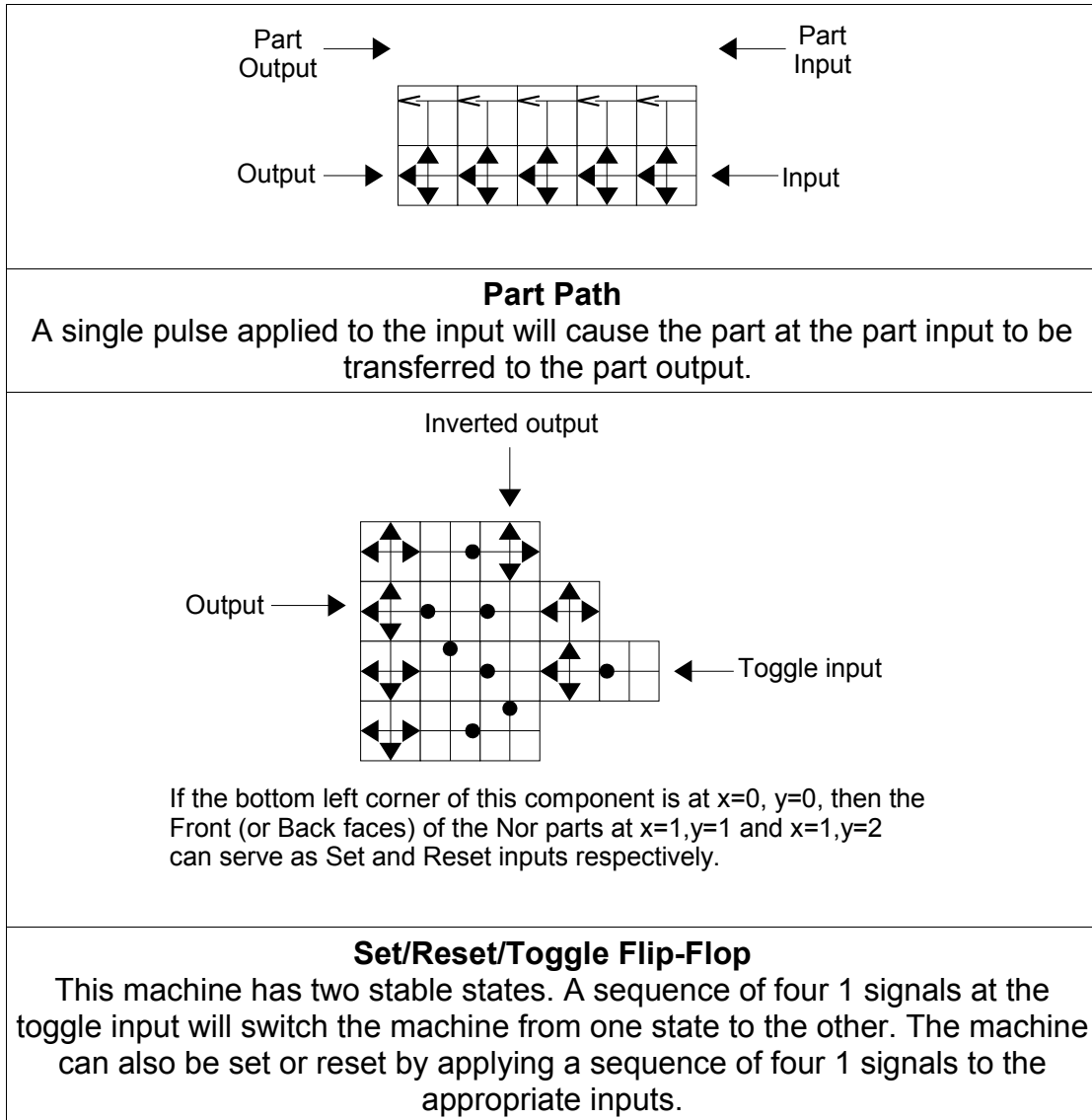


Table 2. Two simple machines in the CBlocks3D environment.

The Delta and Nor parts can be used together to make logic circuits. It can be shown that any computation that can be performed by a Turing machine with a finite tape can be performed by a suitable arrangement of Delta and Nor parts.

### 3. A Programmable Constructor

A programmable constructor has been designed and implemented in the CBlocks3D environment. This constructor contains a sequence of instructions which directs the machine to build other machines, part by part. The machine obtains its parts from a disorganised collection, and is capable of discriminating between parts and storing parts for later use.

Figure 1 shows a schematic diagram of the programmable constructor.

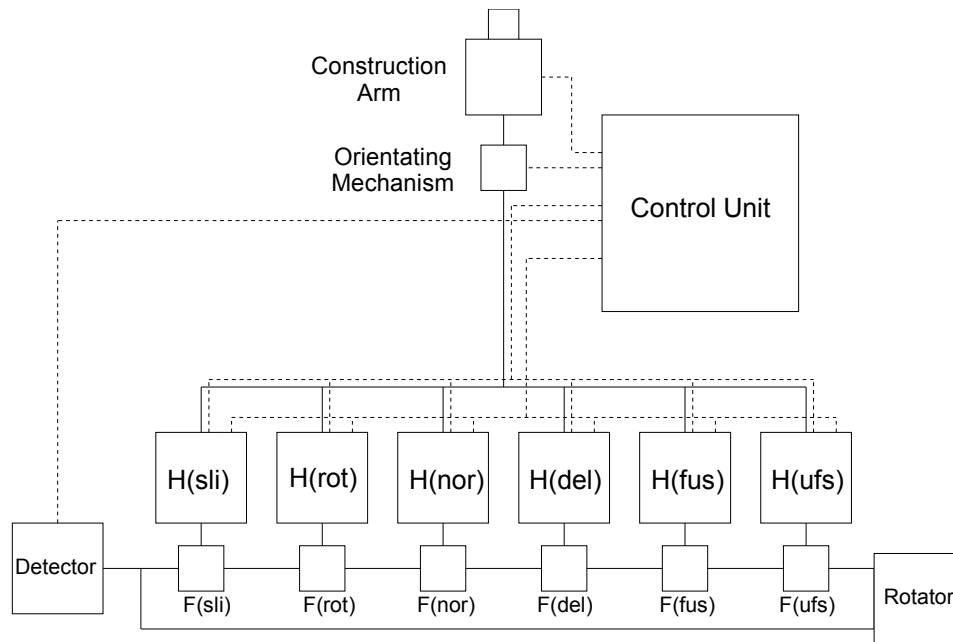


Figure 1. A schematic diagram of the programmable constructor. Three-letter abbreviations are used for parts: sli = slide, rot = rotate, nor = nor, del = delta, fus = fuse, ufs = unfuse.

In Figure 1, dashed lines represent signal paths made from Delta parts (and occasional Nor parts). Solid lines represent paths along which individual parts travel, similar to the 'Part path' in table 2.

The constructor has two operating modes: Gather mode and Construct mode. In Gather mode the constructor moves around its environment encountering parts, classifying them and storing them. In Construct mode the constructor builds machines using parts from its store. The constructor switches from Construct mode to Gather mode when the control unit executes a 'GATHER' instruction. The constructor switches from Gather mode to Construct mode when all of its part storage hoppers are full.

The **Detector** is active when the constructor is in Gather mode, and is responsible for moving the constructor around in its environment and detecting that a part has been encountered. When a part is encountered, the Detector passes the part to the first Filter and waits for the part to be sorted and stored.

**F(x)** are Filters for the 6 different types of part. The Filters are arranged one after another in a line, and parts pass through them one by one. Each filter can detect a single part that is oriented in a subset of the part's behaviourly-distinct orientations. For example, **F(sli)** can detect a slide part in any of the four orientations in which the input to the part is to the North (upside-down compared to the diagram in table 1). When a filter detects a part, the part is diverted to a Hopper (after being placed into a known orientation).

The **Rotator** is a mechanism that will cycle parts through all 24 possible orientations. If a part passes through all filters without being detected, it is rotated by the Rotator and sent back to the first filter to repeat the process.

**H(x)** are Hoppers for the 6 different part types. When a Hopper receives an input part, it joins the part onto the collection of parts that it contains. In response to a specific signal, a Hopper will dispense a part from its collection. When a part is dispensed from a Hopper it will pass through the Orientating Mechanism and then to the Construction Arm.

The **Orientating Mechanism** contains a 5-bit register that is set by an 'ORIENT' instruction. When a part passes through the Orientating Mechanism, it will be set to the orientation specified in the register.

The **Construction Arm** is an arm with a Tip that can be instructed to move to and fro in all 3 dimensions. The Construction Arm will pass a part that it is given to its Tip. The Tip contains 'fuse' parts which will join the part to the machine being constructed.

The **Control Unit** contains memory and logic for executing a sequence of instructions. Figure 2 shows a schematic diagram of the Control Unit.

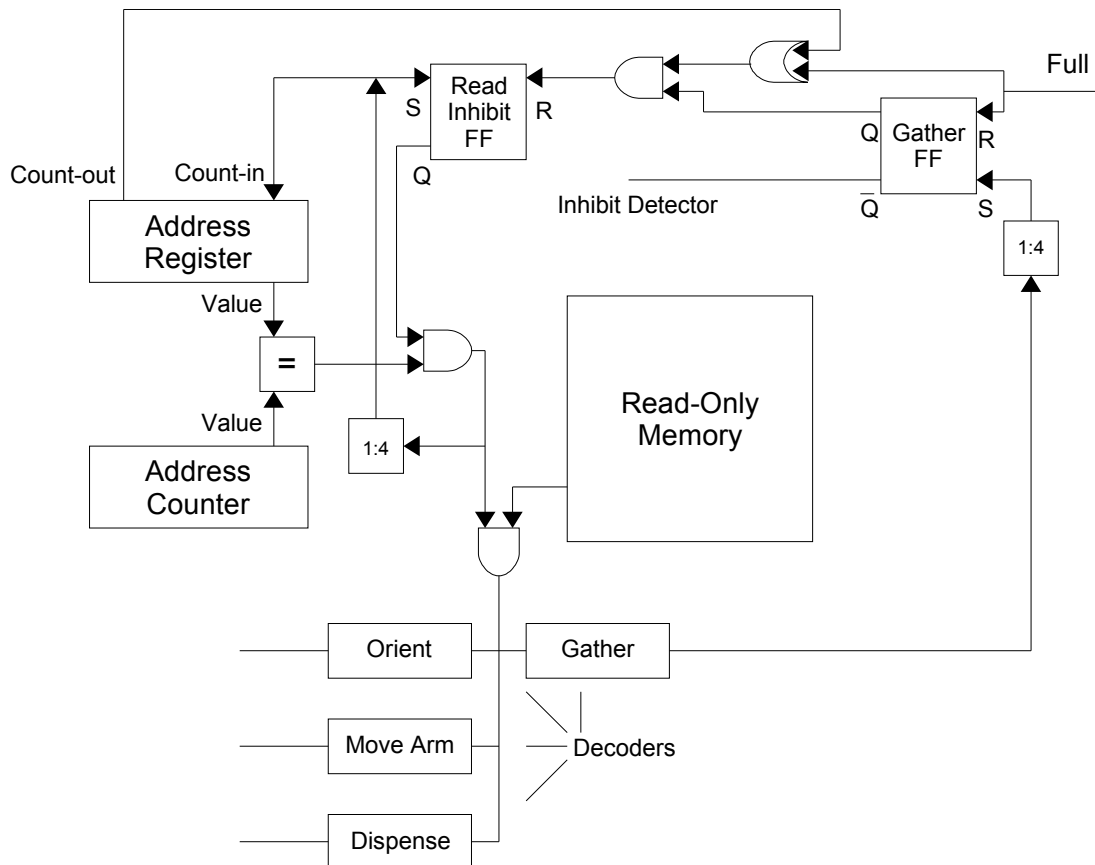


Figure 2. Schematic diagram of the control unit.

The **Read-Only Memory** contains 128 5-bit words. The memory architecture is similar to the delay-line architecture that was used in early electronic computers. Each bit-plane of the memory is a folded loop of 128 Delta parts. A pattern of 128 bits in this loop will cycle continuously around the loop. The loop contains an output tap at a particular point. The **Address Counter** is a 7-

bit counter that increments every time step. Therefore, the Address Counter specifies which of the 128-bits in the memory loop is currently on the output tap. By comparing the Address Counter with the Address Register, a read pulse can be generated when the bit addressed by the Address Register is on the output tap. This pulse can be ANDed with the signal from the output tap to extract the addressed bit from the memory loop.

The **Address Register** is a 7-bit counter that increments by one when a sequence of four '1' pulses is fed into its 'count-in' input.

The components labelled '1:4' in figure 2 are pulse lengtheners, that output a sequence of four '1' pulses in response to a single '1' pulse.

The **Read Inhibit Flip Flop** in figure 2 is used to prevent a false read pulse shortly after an increment signal is applied to the Address Register, when the Address Register passes through transient states. The Read Inhibit Flip Flop is also used to prevent the execution of instructions when the constructor is in Gather mode.

The **Gather Flip Flop** in figure 2 is set when the 'GATHER' instruction is executed, and reset when all of the part storage hoppers are full. This Flip Flop is used to prevent the Read Inhibit Flip Flop from being reset when the constructor is in Gather mode and also to inhibit the detector when the constructor is in Construct mode.

The **Decoders** are used to decode 5-bit instructions. The **Gather Decoder** outputs a single pulse when a 'GATHER' instruction is executed. The **Orient Decoder** responds to an 'ORIENT' instruction by inhibiting all of the other decoders during the next memory-read cycle and then storing the next word retrieved from the memory in a 5-bit register. 5 signal paths from this register go to the Orientating Mechanism shown in figure 1.

The **Dispense** and **'Move Arm'** Decoders respond to 'DISPENSE' and 'MOVE' instructions respectively and both output serially encoded signals. For example, in response to a 'DISPENSE p' instruction, where p is a part and is encoded using the three bits xyz, the Dispense Decoder will output the sequence '1000x0y0z01', and this sequence will be fed to every Hopper. Each Hopper H(p) responds to a different sequence of this form by dispensing a part contained in the Hopper.

Instructions are encoded using 5-bits. The 'ORIENT' instruction requires 2 5-bit words. Table 3 lists the instructions supported by the programmable constructor.



Instruction	Encoding	Notes
NOP	01111	No operation
GATHER	00111	Switch the constructor into Gather mode to start collecting and storing parts. When all of the hoppers are full the constructor will return to Construct mode and continue executing instructions.
ORIENT	10000 xxxxx	xxxxx specifies the orientation.
MOVE	00xxx	xxx specifies the direction in which to move the construction arm: 000 = Back, 001 = North, 010 = Front, 011 = South, 100 = East, 110 = West
DISPENSE	01xxx	xxx specifies the part type to dispense: 000 = slide, 010 = rotate, 011 = delta, 100 = nor, 101 = fuse, 110 = unfuse

Table 3. Instructions supported by the programmable constructor.

Figure 3 is a graphical representation of the programmable constructor. (Compare this to the schematic diagrams in Figures 1 and 2).

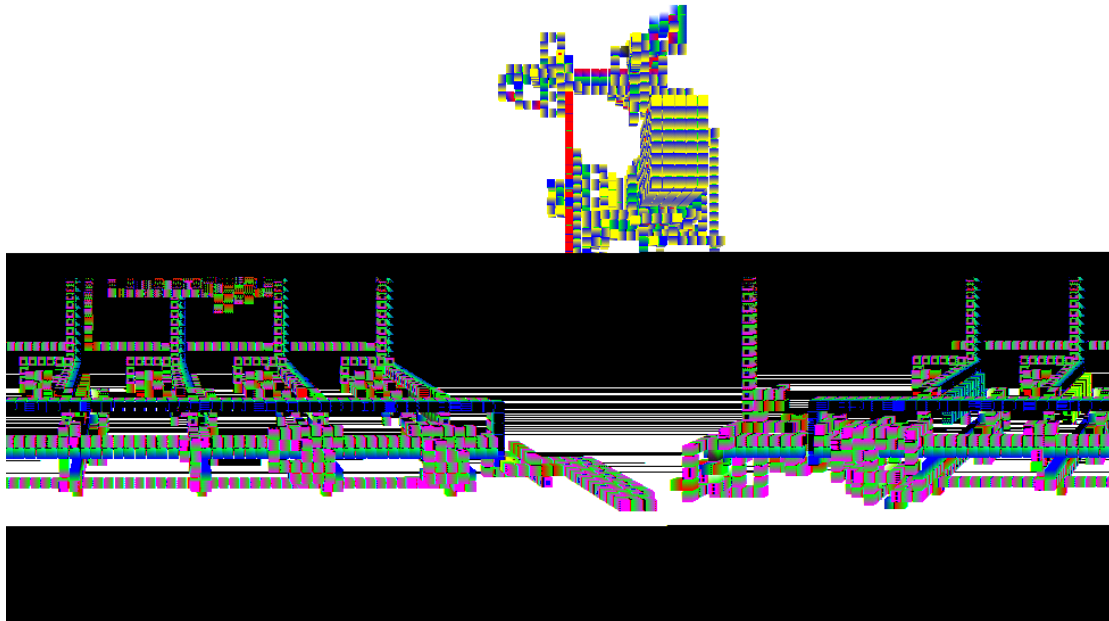


Figure 3. A graphical representation of the programmable constructor.

The constructor is made from 5040 parts, with the following numbers of each type of part:

- 3531 Delta parts (of which 640 make up the Memory) (Yellow)
- 688 Nor parts (Green)
- 676 Slide parts (Red)
- 23 Rotate parts (Cyan)
- 63 Fuse parts (White)

## 59 UnFuse parts (Grey)

These figures include 14 of each part type that are present as initial contents of the constructor's hoppers. Those reading this paper in colour can identify the part types in Figures 3 and 4. The input face of a part is coloured blue, the business end of a part is coloured using the colours given above.

### 4. Example

Before giving an example program for the constructor, it is necessary to describe the Construction Arm in more detail.

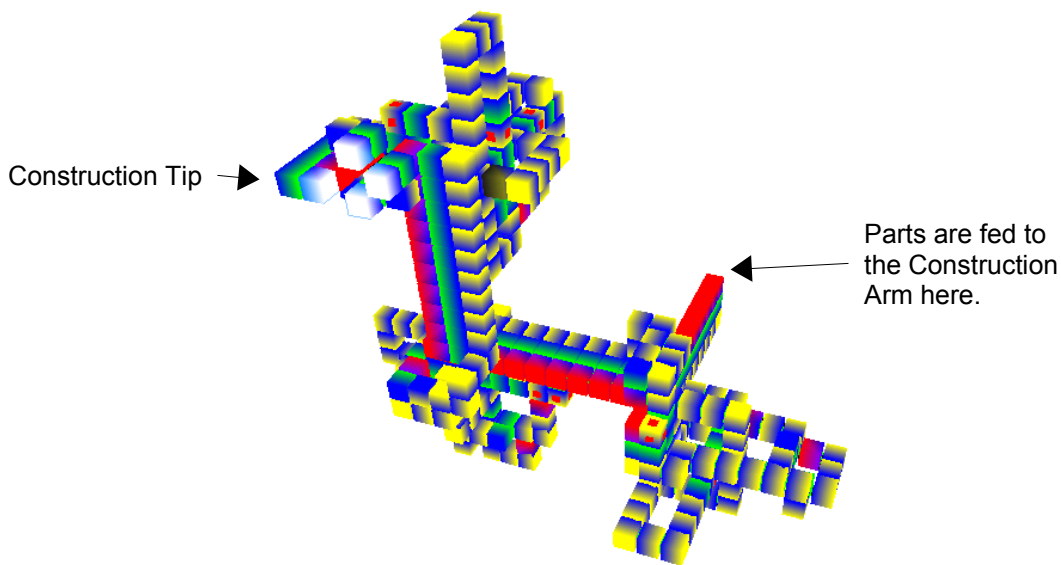


Figure 4. The 3-axis Construction Arm and Construction Tip. This view is from the Back direction, so East is to the left of the page, West is to the right of the page and Front is into the page.

Figure 4 is a graphical representation of the Construction Arm. To the right of the page is the path which connects the Construction Arm to the rest of the programmable constructor, and along which parts are fed to the Construction Arm. Three orthogonal paths can be seen in Figure 4. At the end of the first path and also at the end of the second path (i.e. the Front-Back path and the East-West path) is a mechanism which moves the *adjoining* path back and forth in response to a serially encoded signal. The Construction Tip is attached to the end of the North-South path by a similar movement mechanism.

The Construction Tip can therefore be positioned anywhere within the limits of the arm. The Construction Tip contains four Fuse parts which connect a newly placed part to other parts. A newly placed part will be connected to the part to its north, the part to its south, the part to its west, and the part to its back, if any of these parts exist.

The Fuse part which connects the back of a newly placed part to the part behind protrudes from the Construction Tip so as to be level (in the front-back

direction) with the newly placed part, and lies to the east of the newly placed part. Therefore, construction of machines must proceed in a west-to-east fashion, otherwise this part of the Construction Tip would push the machine under construction along.

Machines can be constructed in layers, with each layer being constructed from west-to-east as described in the previous paragraph. When a layer is complete, the whole construction can be pushed back one unit and the next layer begun. In this way, machines of arbitrary extent along the Front-Back axis can be constructed.

Table 4 gives a sequence of instructions that cause the programmable constructor to construct the Set/Reset/Toggle Flip-Flop shown in table 2.

MOVE West	MOVE East	ORIENT EN	MOVE North
MOVE West	ORIENT EN	DISPENSE Nor	ORIENT WN
MOVE West	DISPENSE Nor	NOP	DISPENSE Nor
MOVE West	NOP	MOVE West	NOP
ORIENT NE	MOVE North	MOVE East	MOVE East
DISPENSE Del	ORIENT NE	DISPENSE Del	DISPENSE Del
NOP	DISPENSE Nor	NOP	NOP
MOVE South	NOP	MOVE South	MOVE East
ORIENT WN	MOVE North	MOVE East	DISPENSE Nor
DISPENSE Del	MOVE East	MOVE East	NOP
NOP	ORIENT WN	ORIENT NE	MOVE South
MOVE South	DISPENSE Nor	DISPENSE Del	MOVE West
ORIENT SE	NOP	NOP	MOVE West
DISPENSE Del	MOVE East	MOVE South	MOVE West
NOP	DISPENSE Nor	MOVE West	
MOVE South	NOP	MOVE South	
DISPENSE Del	MOVE North	DISPENSE Nor	
NOP	MOVE West	NOP	

Table 4. A sequence of 68 instructions (encoded in 77 5-bit words) used to construct a Set/Reset/Toggle Flip-Flop

Notice that every 'ORIENT' instruction is followed by two letters. These letters represent the value that the register in the Orientating Mechanism is to be set to, and therefore the orientation that any part passing through the Orientating Mechanism will be given. The first letter is the direction in which the part's main axis is to point. The second letter gives the orientation of the part around its main axis.

Notice also that a 'NOP' instruction is executed after every 'DISPENSE' instruction. This is because it can take longer than a single instruction cycle for a part to be dispensed from a hopper and passed to the Tip of the Construction Arm.

## 5. Self-Replication

### 5.1 Requirements for Self-Replication

If the constructor can be developed into a self-replicating system, the following requirements must be met:

a. The constructional capability of the Construction Arm must be such that all of the subsystems of the constructor can be constructed and then assembled.

There is no limit on the spatial arrangement of the parts that the Construction Arm places within its domain of operation. However, there are limits on the connectivity state that the construction arm can give to its constructions. In addition, where a construction needs to be set to an initial state, measures may have to be taken to ensure that this state is set. The example construction in section 4 illustrates the second point.

b. The domain of operation of the Construction Arm must be large enough to allow a replica constructor to be constructed.

One way to achieve this is firstly to ensure that the dimensions of the constructor are such that  $FB < NS < EW$ , where FB is the Front-Back span of the constructor, NS is the North-South span of the replicator and EW is the East-West span of the constructor. Secondly, the East-West reach of the Construction Arm must be greater than or equal to NS, and the North-South reach of the Construction Arm must be greater than or equal to FB. When a parent constructor builds a child, the child is oriented as in figure 5, so that the East-West and North-South spans of the child fit within the East-West and North-South spans of the parent. This is the same as the arrangement of parent and child in the 'extruding brick' architecture described in section 4.11.3 of [16].

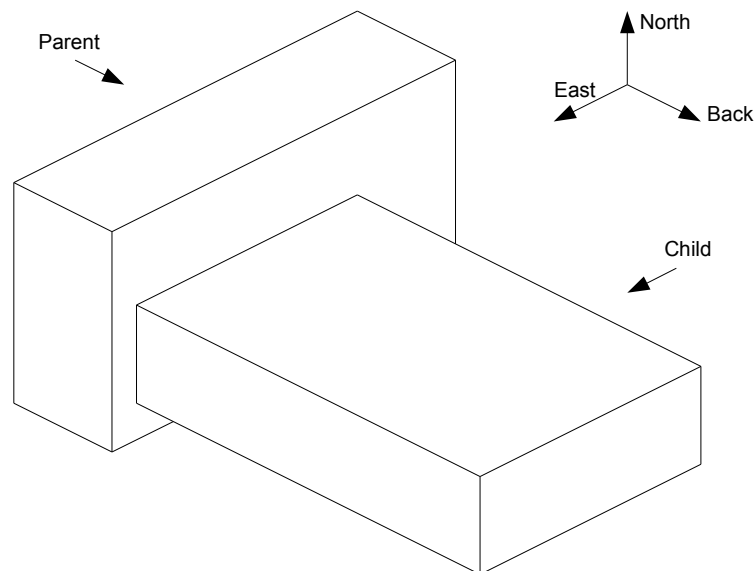


Figure 5. Orientation of parent and child that allows the child to be extruded from the parent constructor.

c. The description of the self-replicating constructor S (which includes memory M as a subsystem) must fit within the memory M.

For this to be possible, a self-replicating machine derived from the programmable constructor presented in this paper according to the outline given in this section must support looping or subroutine calls. This is because

every part of a construction requires at least two 5-bit word instructions (one instruction to move the Construction Arm and one instruction to dispense the part) occupying ten Delta parts within the memory. Therefore the memory M cannot contain enough instructions to construct a copy of M (let alone the rest of S) unless instructions can be repeated or re-used.

## 5.2 Modifications for Self-Replication

To develop the constructor into a self-replicating system requires the following modifications and additions (in addition to those mentioned above):

- The memory must be enlarged.
- A copying mechanism must be added to the memory so that the sequence of instructions used to direct replication can be passed from parent to child after replication is complete.
- Mechanisms must be added that reject a part if the hopper for the part is full.

The performance of a self-replicating system derived from the programmable constructor could be improved by making the following changes:

- The sorter could be modified so as to be able to deal with more than one part at a time.
- The subsystems from which the constructor is built could be scrutinized with a view to reducing the part count.
- The control unit could be modified to support both looping *and* subroutine calls. (It is a logical requirement that either looping or subroutine calls be supported, but for efficiency reasons it is desirable to support both).

## 6. Conclusion

A programmable constructor has been demonstrated in an environment that has some properties in common with the cellular automaton environments that have previously been used to study construction and self-replication. Unlike cellular automaton environments, the CBlocks3D environment supports motion, and so problems to do with the transportation, management and placement of parts can be addressed in the CBlocks3D environment.

Only six simple part types are needed to make a programmable constructor in CBlocks3D. This is a considerable reduction compared with the 24 part types used in Cblocks [15] and the 22 part types used in Nodes [14]. The six part types can be classified into three groups. Firstly, there are parts that enable a machine to make use of the kinematic features of the CBlocks3D environment (i.e. motion and physical connection): Slide, Fuse and Unfuse parts. Secondly, there are parts that propagate and process signals: Delta and Nor parts. Thirdly, there is the Rotate part, which compensates for the fact that the CBlocks3D environment does not have laws of motion that permit parts to rotate.

The decision about which functions to choose for basic part types was determined both by the characteristics of the CBlocks3D environment, and by a need to have as small a set of part types as possible so as to minimize the size of the sorting and storage mechanisms of the programmable constructor. One natural question that arises when considering the choice of basic part types is the question of whether the set of part types can be reduced further. The Delta and Nor parts could be combined into a single part that both propagates and processes signals. Alternatively, mechanical logic could be used and signal processing parts could be done away with altogether (as described below) but these seem to be the only simplifications that can be made.

However, it is reasonable to suggest that changing the laws of the simulation environment would change the range of part types needed. For example, a continuous space environment like that used in the Nodes environment permits machines to rotate, so the Rotate part may be made redundant. An environment supporting force fields or snap-fit connectors might make the Fuse and UnFuse parts redundant if connections between parts could be made and broken by pushing and stretching.

The need for dedicated signal propagation and signal processing elements could be removed by using mechanical logic. It can be shown that an environment like CBlocks3D supporting only a single part type called S (S is like the Slide part described in this paper, but with no input and continually active) can be used to implement a machine that can perform any computation that can be performed by a Turing machine with a finite tape.

The work described in this paper leads to the following questions:

1. Can the programmable constructor described here be developed into a self-replicating system? (i.e. Can requirements **a**, **b** and **c** be met?)
2. Can a constructing automaton be made in an environment like CBlocks3D using the single part type S if the environment is modified so that connection, disconnection and rotation result from opposing or orthogonal forces applied to a part?
3. Can a constructing automaton or a computing automaton be made in a continuous space environment supporting a part like S with a suitable force field between neighbouring parts?

## **7. References**

1. K. Eric Drexler, Engines of Creation: The Coming Era of Nanotechnology, Anchor Press/Doubleday, New York, 1986. <http://www.foresight.org/EOC>
2. John von Neumann, Theory of Self-Reproducing Automata, A.W. Burks, ed., University of Illinois Press, Urbana, Illinois, 1966. Part 1, Lecture 5.

3. According to John R. Koza, "Artificial Life: Spontaneous Emergence of Self-Replicating and Evolutionary Self-Improving Computer Programs", *Artificial Life III*, Addison-Wesley, Reading, Mass., 1994. pp 225-262.
4. E.F. Codd, *Cellular Automata*, Academic Press, New York, 1968.
5. Christopher G. Langton, "Self-reproduction in cellular automata," *Physica D* 10, 1984. pp 135-144.
6. Barry McMullin, John von Neumann and the Evolutionary Growth of Complexity : Looking Backwards, Looking Forwards..., *Artificial Life VII*, MIT Press, Cambridge Mass. 2000. pp 467-476
7. L.S. Penrose, R. Penrose, "A self-reproducing analogue," *Nature* 179, 8 June 1957. p 1183.
8. Harold J. Morowitz, "A model of reproduction," *American Scientist* 47, June 1959. pp 261-263.
9. Jackrit Suthakorn, Andrew B. Cushing, Gregory S. Chirikjian, "An autonomous self-replicating robotic system", *Proc. 2003 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM 2003)*, Kobe, Japan, 2003.
10. Matt Moses, "A Physical Prototype of a Self-Replicating Universal Constructor", Master's Thesis, Department of Mechanical Engineering, University of New Mexico, 2001.  
<http://www.home.earthlink.net/~mmoses152/selfrep.doc>
11. Tihamer Toth-Fejel, "Modeling Kinematic Cellular Automata: An Approach to Self-Replication", NASA Institute for Advanced Concepts, 2004.  
[http://www.niac.usra.edu/files/studies/final\\_report/pdf/883Toth-Fejel.pdf](http://www.niac.usra.edu/files/studies/final_report/pdf/883Toth-Fejel.pdf)
12. S. Murata, E. Yoshida, A. Kamimura, H. Kurokawa, K. Tominata, S. Kokaji, "M-TRAN: Self-Reconfigurable Modular Robotic System", *IEEE/ASME Transactions on Mechatronics*, Vol 7, No. 4, 2002, pp 431-441.
13. V. Zykov, E. Mytilinaios, B. Adams, H. Lipson, *Self-Reproducing Machines*, *Nature*, Vol. 435 No 7038, 2005, pp 163-164.
14. William M. Stevens, *An Environment for Simulating Kinematic Self-Replicating Machines*, *Artificial Life IX*, MIT Press, Cambridge, Mass. 2004. pp 39-44.
15. William M. Stevens. Unpublished work on CBlocks simulation environment, 2005. <http://www.srm.org.uk>
16. Robert A. Freitas and Ralph C. Merkle, *Kinematic Self-Replicating Machines*, Landes Bioscience, Georgetown, Texas, 2004.
17. Michael A. Arbib, *Theories of Abstract Automata*, Prentice-Hall, Englewood Cliffs, NJ, 1969. Chapter 10. 'Machines Which Compute and Construct'
18. Richard L. Thompson and Narendra S. Goel, *Movable Finite Automata (MFA) Models for Biological Systems I: Bacteriophage Assembly and Operation*, *Journal of Theoretical Biology*, Academic Press, No. 131, 1988. pp 351-385.