Robotica (2011) volume 29, pp. 153–176. © Cambridge University Press 2011
doi:10.1017/S0263574710000688

A self-replicating programmable constructor in a kinematic simulation environment

William M Stevens*

Department of Physics and Astronomy, Open University, Milton Keynes MK7 6AA, UK

(Received in Final Form: October 25, 2010)

SUMMARY
Cellular automata can model the information processing aspects of a self-replicating programmable constructing system (SRPC), but these models do not contain any notion of material parts and are poor at modelling features of a system that depend upon the motion and connectivity of its components. Physical systems with thousands of parts have the disadvantage that they are time consuming to develop. To overcome these limitations, a simulation environment is presented at a level of abstraction that models motion and connectivity. An SRPC which takes a disorganised collection of parts as its input has been implemented in this environment.

KEYWORDS: self-replication, simulation.

1. Introduction

There are a number of open problems related to self-replicating systems. One group of problems relates to the emergence and evolution of self-replicating systems:

(1) What are the conditions that must be satisfied for a system to demonstrate open-ended evolutionary change?
(2) How did the first natural self-replicating systems arise on Earth?
(3) How did the earliest living organisms acquire the organised structure and the information processing architecture that they now possess?

These questions are among those that the fields of Artificial Life and Theoretical Biology seek to answer.[1,2] A second group of problems relates to the design and engineering of artificial self-replicating machines:

(1) Can a robot be made that can construct other machines, including a replica of itself, in an environment containing a collection of passive component parts?
(2) Extending the above question, can such a robot be made to operate in an environment containing only materials that occur naturally in that environment such as metal ores, other minerals, water and atmospheric gases?
(3) Is there any limit imposed by the laws of physics on the scale at which this type of robot can operate?

This paper addresses the first question from the second group of problems. Naturally these two groups of problems, the questions within them and the techniques used to answer them are not distinct and contain many overlaps. So in attempting to answer one question, it should not be surprising that from time to time other questions are touched upon.

For example, suppose that we wanted to design an automated self-replicating factory to send to the Moon to produce various items that would be useful to future human lunar colonists using nothing but lunar raw materials.[3] To be confident that the factory would carry out the task that it was designed for, we would need to be sure that the factory would not undergo evolutionary change.

* Corresponding author. E-mail: william@stevens93.fsnet.co.uk

2. Previous Research into Self-Replicating Programmable Constructing Machines

A number of reviews of research into self-replicating systems have been published. The earliest is that of Laing, which appears in Section 5.2 of Freitas and Gilbreath's report on a 1980 NASA study of an application of self-replicating systems for space missions.[3] Sipper[4] completed a 50-year review in 1998 which concentrated on cellular automata models of self-replication but touched briefly on other models. Freitas and Merkle's 2004 book 'Kinematic Self-Replicating Machines'[5] contains a comprehensive review section covering all kinds of self-replicating systems but in particular focusing on physical self-replicating systems.

This brief review has some overlap with these earlier reviews but a different and more specific focus. The focus is on self-replicating programmable constructors (SRPCs)—machines that can be programmed to construct other machines, and which can construct a duplicate machine as a special case. Characteristic examples of other self-replicating systems will be described, but only to point out how they differ from SRPCs, or how they could be extended so as to become more like SRPCs.

2.1. Self-replication in living systems

In the cell, genetic material in the form of DNA contains information which directs the machinery of the cell to construct all of the cell's constituent parts, including components which play a role in interpreting and processing information from the DNA, structural components which give the cell its shape, and components which work together to control the metabolic activity of the cell. The cell can obtain energy and raw materials from simple molecules in its environment. When conditions are such that the cell is ready to replicate, some of the cellular machinery duplicates

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

the genetic material, the cell grows, and then it divides in two. A mechanism exists to ensure that each half of the division contains a copy of the genetic material. Thus, the cell becomes two new cells.[6]

2.2. Von Neumann's work

In the late 1940s, von Neumann set out to answer several questions surrounding machines capable of constructing other machines.[7] Von Neumann asked whether it is possible for a single automaton in a given environment to be construction-universal within that environment, i.e. able to construct every other automaton, if given a suitable description. He then asked whether an automaton of this type could construct another automaton exactly like itself.

Von Neumann answered these questions by devising a 29-state cellular automaton (CA) with transition rules designed to support five types of component part which allow the implementation of signalling pathways and logical operations, as well as construction operations. One type of component, the Unexcitable component, is conceptually similar to empty space. In the context of von Neumann's CA, construction operations are those which result in the Unexcitable component being transformed into other components, or that turn a component back into the Unexcitable component.

Von Neumann did not fully specify a cell-by-cell description of a design for an SRPC. Thatcher[14] went a long way towards doing this, but only recently has this been completed.[8–10]

Freitas and Merkle[5] point out some of the limitations of von Neumann's CA model from the perspective of physical realisation. These are paraphrased below:

(1) The system does not directly permit the movement of objects in space. In von Neumann's environment motion is a complex process of deleting cell states at one location and recreating them at another.
(2) The system is synchronous: All state transitions in cells occur simultaneously.
(3) The system provides only a limited capacity to detect the states of cells, making self-examination and self-repair difficult.
(4) Related to 3, the region in which a new machine is to be created must be entirely empty, otherwise construction cannot proceed.
(5) The environment is essentially an information processing environment and does not attempt to model manipulation and transformation of physical materials.

Von Neumann's design is complex. Since von Neumann's time several very simple self-replicating systems have been devised (for example, see Sections 2.5–2.7). This has led to a debate about what it was that von Neumann was trying to achieve in his work and why his design is so complex. Von Neumann was well aware that there are some simple systems, such as growing crystals, that exhibit a kind of self-replicating behaviour, and he proposed a definition of self-replication that included the ability to undergo inheritable mutations as well as the ability to make another organism like the original.[11]

McMullin[12] argues that the problem that von Neumann solved was primarily the problem of how machines can construct other machines, and in particular how machines can give rise to machines no less complex than themselves, which was an unsolved problem in nature at the time. McMullin points out that von Neumann's CA pattern that can be programmed to construct any other pattern, including a duplicate pattern, is a solution to that problem. Simpler self-replicating systems, for example Langton's self-replicating loop,[13] do not solve the same problem because these systems cannot construct other systems.

The debate about the relevance and purpose of von Neumann's work highlights the fact that the capacity for self-replication is one design requirement or constraint among many others that a system may have. Any two systems that have self-replicability as one of their requirements will not necessarily have other requirements in common, and it may not be meaningful to compare them on the basis of this single aspect of their design.

2.3. Thatcher's work

Thatcher developed an alternative strategy for self-replication in von Neumann's CA environment.[14] Von Neumann's self-replicating automaton contained a subsystem dedicated to copying the instruction tape from a parent automaton to its child, and the controlling subsystem in von Neumann's automaton was not a universal computer. Thatcher realised that if a universal computer is used to control the automaton, then the self-describing property of this computer[15] will make the instruction-tape-copying subsystem superfluous.

From a theoretical point of view, Thatcher's development is very elegant. It reduces the amount of technical detail that is needed to prove that a particular environment supports SRPCs. If it can be shown that a universal constructing system controlled by a universal computer can be made in a particular environment, then it follows automatically that the system will be capable of self-replication in that environment.

2.4. Laing's kinematic automaton system

In the late 1970s, Laing described a range of plausible architectures for self-replicating systems made from abstract modular parts capable of interacting logically and kinematically with other parts.[16]

Laing first of all discussed the architecture used by von Neumann, in which a description of a self-replicating automaton is first interpreted by a constructor which builds a replica of the automaton, and then duplicated and a copy of the description given to the replica. He then described a number of other architectures in which a machine does not need to contain any kind of description of itself. For example, he showed that it is possible for a machine to examine itself and construct a replica using information obtained during this examination.

2.5. Template based systems

If no other requirement is given for the design of a self-replicating system except that it be able to make another system like itself, then designing such a system is trivially

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

easy in a CA environment and not very difficult in a physical environment.

If one adds the requirement that the system must be capable of transferring information from parent system to child system, then the problem becomes more complex. In the case of a CA environment, the increase in complexity is not great: one simply defines a state space large enough to accommodate the number of bits to be transferred. In the physical world however, this additional requirement does introduce some more complexity.

This problem was first addressed by Lionel and Roger Penrose.[17] Lionel Penrose's speculations about simple self-replicating systems in biology led him to propose a template-based model for replicating a linear structure composed of a number of template subunits.[18] Penrose envisaged an environment containing a large supply of individual subunits moving about in a random fashion in the same manner as molecules in a gas. A linear template structure in this environment would induce individual subunits to form a replica structure parallel to and in direct contact with itself, one unit at a time. The replica structure would then separate from the template when it was complete. This scheme has subsequently been realised by Griffith *et al.*[19] Smith *et al.*[20] and Hutton[21] have developed simulation models of template-based replication.

2.6. Path-directed self-replicating robots

Using LEGO® modules, Chirikjian's research group built a number of simple self-replicating robots.[22] In all of these systems, a robot is able to bring together a small number of separate modules and assemble them into a replica of itself. Information that controls the robot's movements exists in the form of a path drawn on the ground, which a path-following mechanism in the robot tracks. The robot does not produce a duplicate path, so this part of the system is not replicated.

2.7. Langton's self-replicating loops

Langton[13] devised a simple self-replicating loop structure embedded in an 8-state CA. The loop structure contains a 'genotype' in the form of a sequence of instructions continuously circulating within the loop, a 'construction process' in the form of a sequence of state transitions that take place at a growing tip of the replicating loop structure and a 'phenotype' in the form of the protective sheath of the loop.

2.8. Programmable constructors based on simple self-replicating systems

The self-replicating systems described in Sections 2.5–2.7 are not designed to be capable of constructing anything except copies of themselves. Nevertheless it is still worth considering whether such systems can inform the study of SRPCs. The most obvious consideration is that life as we know it today arose from a much simpler self-replicating system and probably from a system so simple that it occurs spontaneously in the right conditions. There are several different hypotheses about the origin of life, but in each one there is a continuous evolutionary path from very simple self-replicating systems with little constructional capability to the diverse range of living systems that we see today. Today's organisms are able to make teeth, cameras, complex information processing systems, oil, potatoes and medicines.

So perhaps one way to make artificial SRPCs is to mimic the path taken by evolution: begin by investigating simple self-replicating systems and then gradually develop and modify them so as to extend their programmability and their constructional capabilities. This approach has been taken by Ewaschuk and Turney where a simple template-based replicating system[20] is later augmented by a templating-folding system so that arbitrary shapes can be constructed.[23] Tempesti[24] also adopted this approach by showing that Langton's loop can be augmented with additional rules which lead to a greater constructional capability.

Although nature begins with simple systems that over time become more complex, nature does not do this for any reason of efficiency or because it is a good process to follow to end up with the desired outcome. Nature does this because it has no other alternative.

So although we may draw inspiration from nature we should not assume that nature does things in the best possible way, or that living organisms represent any ultimate limit on the kind of self-replicating constructing machines we may create, even if we limit ourselves to organic elements.

2.9. Systems derived from living systems

Advances in the field of synthetic biology have led to the ability to perform genetic engineering at the genome level.[25] This makes it possible to engineer cells to contain a metabolism designed for a specific purpose and a genome containing only information that is essential for the cell to function and to carry out the task that it is designed for.

The disadvantages and difficulties of this approach stem from the fact that cells are evolved machines; one of the primary constraints that affects the architecture and programmability of living things is that they must be capable of evolution. Evolutionary capacity requires the ability to change and the ability to continue working despite the change and produce offspring that inherit the change. For many applications of self-replicating systems, these abilities are highly undesirable. We usually do not wish designed machines to change by themselves; this is normally counted as a malfunction. If a machine does change or malfunction, then we would like the machine to be able to fix or work around the problem rather than pass the malfunction on to other machines.

One of the consequences of synthetic biology may be that synthetic cells will gradually lose their ability to undergo natural evolution as their architecture is modified to make them more easily programmable by humans.

2.10. Von Neumann's kinematic model

Before developing his 29-state CA for exploring construction and self-replication von Neumann envisaged a physical system in which he thought it might be possible to devise a constructing machine, quoted below from page 82 of Theory of Self-Reproducing Automata:[7]

> The constructing automaton floats on a surface, surrounded by an unlimited supply of parts. The constructing automaton contains in its memory a

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

description of the automaton to be constructed. Operating under the direction of this description, it picks up the parts it needs and assembles them into the desired automaton.

Burks refers to this system as the *kinematic model*. Von Neumann went some way towards considering the types of part the kinematic model should support and came up with a list of eight types. On page 81 of Theory of Self-Reproducing Automata,[7] Burks attempted to reconstruct the most detailed description of these parts that von Neumann gave, summarised below:

- Four of the parts perform logical and information processing operations.
- The fifth part is a *rigid member*, from which a rigid frame for an automaton can be constructed.
- Connections are made by a *fusing organ* which, when stimulated, welds or solders two parts together.
- Connections may be broken by a *cutting organ* which, when stimulated, unsolders a connection.
- The eighth part is a *muscle*, used to produce motion.

Von Neumann postponed further work on this model after deciding that the CA model would be easier to reason about and would be sufficient for answering the questions that he was interested in at that time. Burks[26] believed that pursuing a kinematic model was unnecessary for the study of constructing automata and yet there are at least two subsystems of von Neumann's automaton that can be expressed more naturally in a system that supports motion. The first of these is the instruction tape. To access a particular location on the tape, a path must be extended to that location so that it can be read and then a signal from that location must be fed back to the main body of the constructor. This is very time consuming and accounts for most of the execution time of the machine. If the system supported motion, then the instruction tape could be moved relative to the constructor body and its contents accessed more quickly.

The second and most obvious subsystem that would benefit from a model of motion is the construction arm. In a similar way to the tape reading mechanism, the construction arm moves by laboriously erasing itself from one location and recreating itself in another. So we see that despite trying to develop a model that does not support any concept of motion, motion returns again by the back door in an ill-fitting way.

Burks also stated that any kinematic model would be too remote from chemistry, physics and mechanics to be of much interest in its own right, but this overlooks the fact that science contains many models that are remote from fundamental physical laws and that it is precisely for this reason that they give insight. A very good example of how kinematic interactions result in very similar behaviour in systems where the physical, mechanical and chemical laws are very different in each case is that of template-based replicating systems.

The systems of Penrose, Griffith *et al.*, Hutton and Smith *et al.* all have different fundamental laws of motion and interaction—the first two being physical systems in different media and the latter two being discrete space and continuous space computer simulations, respectively. What these systems have in common is that they implement concepts of motion, proximity and connectedness at a higher level of abstraction and as a result very similar behaviour can be observed in all of these systems.

It is true that the design of a kinematic model will require some arbitrary choices about which concepts to model and how precisely to model them but similar choices have to be made for logical models: which logical primitives should be used? Which model of computation should be implemented?

Furthermore, a kinematic model can answer questions that a logical model cannot answer. Von Neumann showed that it is possible to devise an SRPC in an abstract logical environment. This tells us that there is no inherent impossibility in the control, communication and information processing aspects of the design of an artificial physical SRPC but it does not tell us whether there are any other barriers to building such a machine. For example, we might find that when it comes to designing parts for such a machine there is some minimum complexity that parts must have in order for the system to be closed with respect to the identification and classification of parts.

2.11. Moses' programmable constructor

Matt Moses developed a physical constructing system designed to be capable of constructing a replica of itself under the control of a human operator.[27] The system is based around a set of 11 different types of tailor-made plastic blocks which can be used to build a controllable manipulator that can pick up other blocks one at a time, position them in three dimensions and then snap-fit them into a structure being built.

The constructing machine is supplied with parts by the operator. Each part is placed in a storage site as it is needed by the machine. The machine moves its manipulator so as to pick up parts from this site, move them into the construction area and place them at the correct location, snapping a part onto the handle of another part as necessary.

Moses describes several limitations of his work, some of which are listed below:

1. The physical implementation of his design did not work as well as expected.
2. The system required external control so was not fully self-replicating.
3. Replacing the human operator with an automatic controller would not be a straightforward matter, since the human operator relies on visual feedback.
4. The set of parts contains considerable redundancy and the choice of parts is somewhat arbitrary.

2.12. Self-replicating modular robots

Considerable work has been carried out on modular robotic systems in which several identical modules (each containing a microcontroller and communications channels and each capable of a small range of movements) are connected together to make a single larger robot capable of a wider range of movements, including locomotion and often manipulation, especially of other robot modules. Freitas and Merkle give an overview of many such systems in Section 3.8 of Kinematic Self-Replicating Machines.[5] Such robots are often designed to be autonomously reconfigurable, in which case the robot

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

is capable of disconnecting, rearranging and reconnecting its modules.

Although not all such systems are designed to be self-replicating systems, there is a sense in which many of them are. In an environment containing a number of disconnected modules, a modular robot may be capable of moving around in the environment and picking up disconnected modules until it has enough to build a replica of itself. The program within the original robot is then transferred to the replica.

Zykov *et al.*[28] make this explicit by building a modular robotic system in which a configuration of four modules can construct a replica configuration when provided with a supply of additional modules in a location known to the robot. A notable feature of the system of Zykov *et al.* is that a single cubic module has only one degree of freedom: it may swivel about a diagonal axis of the cube. Collections of connected modules are capable of exhibiting complex motion as a result of the collective motion of several individual modules. An obvious limitation of this system is that the modular parts are highly complex, considerably more complex than the parts used by Moses.

2.13. The RepRap project

Bowyer *et al.*[29] have developed a rapid prototyping system based around a three-dimensional (3D) printer that is capable of being programmed to manufacture 3D plastic objects. Bowyer and his collaborators hope to enable the system to make as many as possible of its own parts by a two-pronged approach of extending the range of materials that the system can work with (to include, for example, conductive materials that can be used for making circuit boards) so that it can manufacture a wider range of parts and also by reducing the complexity of the parts that the system is built from.

As of 2009, the system is capable of being programmed to build all of its own plastic parts. There are no plans to add assembling capabilities to the system, so although it may ultimately be capable of making most if not all of its own parts, it will not be able to assemble those parts together.

2.14. NASA 1980 summer study on self-replicating systems

Self-replicating systems offer enormous promise for the exploration of space. Indeed without exploiting an artifical self-replicating system of *some* form (for example, a growing, self-sufficient, space-based colony with human beings as an essential component) the long-term exploration of the galaxy is not possible.

Several people have written about this prospect. The most detailed study of self-replicating systems for space exploration was a 1980 NASA Summer Study[3] investigating the possibility of constructing a self-replicating factory on the moon. Such a factory would obtain raw materials from the lunar soil by a variety of chemical processes, then process these materials to make simple parts. The simple parts would then be assembled into subsystems and then the subsystems would be assembled into replica factories.

Section 5.6.4 of the study report[3] describes an outline proposal for a feasibility demonstration system consisting of a robot built from off-the-shelf electronic and mechanical components able to build a replica of itself by picking components from a stockroom and assembling them according to a specified plan.

2.15. Drexler's assembler

In Engines of Creation,[30] K. Eric Drexler describes what he calls an 'Assembler'. This hypothetical machine is capable of operating at the atomic scale, effectively building objects in which every atom is in a specified position.

As originally proposed by Drexler, an assembler consists of a programmable computer and a moveable manipulator arm with a set of interchangeable reaction tips. Between them the tips are capable of performing all of the chemical reactions needed to enable the assembler to construct whatever it is required to construct.

Drexler hypothesises that as a result of this level of construction precision the properties and behaviours of manufactured objects will be greatly extended. In his earlier works on the subject Drexler proposes that in order to make macroscale objects in a short time such assemblers could be given the capacity for self-replication so that in order to make a large object an assembler would first of all generate many more assemblers, which would then cooperate to make the object. Drexler points out that it is not unreasonable to expect that man-made self-replicating molecular manufacturing systems might have greater capabilities in this role than living things. In his later works,[31] Drexler argues that nanoscale self-replicating assemblers are not necessary in order to realise the benefits of molecular manufacturing.

2.16. Simulating self-replicating machines

This review shows that while considerable work has been done on simulating CA based models of self-replicating programmable constructing systems and some work has been done on building simple physical self-replicating machines of various kinds, very little work has been undertaken to model or simulate physical self-replicating programmable constructing machines made from simple component parts.

Each of the self-replicating systems described in this section can be viewed as involving a trade off between a number of different factors including part simplicity, constructional capability and physical realism.

This can be represented by the diagram in Fig. 1 in which several of these self-replicating systems are positioned relative to other systems in such a way as to show how they differ qualitatively from each other in the simplicity of their parts, the range of things that they can construct, and how physically realistic they are.

The region labelled G in Fig. 1 represents the class of fully autonomous physical self-replicating systems whose members can take in very simple parts or raw materials and from them can construct a wide range of other machines in addition to a replica machine. Such a system has not yet been demonstrated.

There are several different ways to progress from existing systems towards G. The simplest living cells already operate on low-complexity materials (i.e. simple molecules) and could perhaps be engineered so as to increase the range of things that they can make. Another approach is to begin with a physical system such as a 3D printer used in the RepRap project, and then work out how to increase the range of things

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

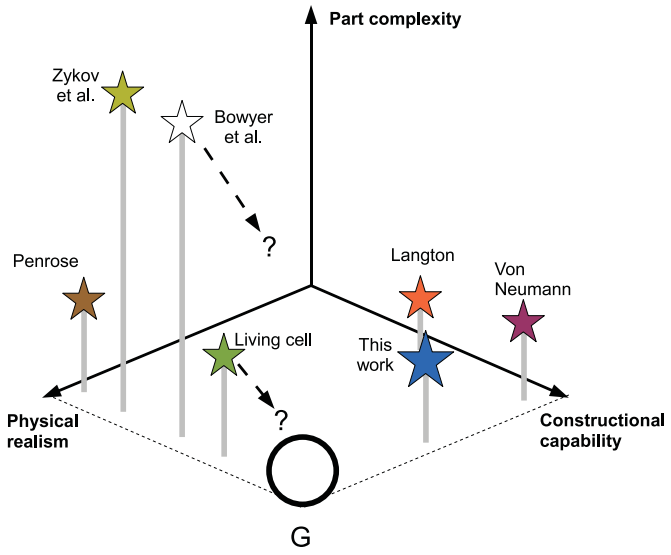


Fig. 1. Map of some existing work on self-replicating systems.

that the system can make. This will have a knock-on effect of reducing the complexity of the set of parts that must be supplied to the system, because it will be able to make more of its own parts from scratch.

The approach towards G that is taken in this paper is to investigate the largely unexplored middle ground between simulations of CA based self-replicating programmable constructing systems made from simple but highly abstract parts and the various physical self-replicating systems that have been constructed.

It is conceivable that region G is not reachable. It could be that machines in this region are impossible to design and are not possible even in principle. Or it could be that machines in this region are possible in principle, but not constructable using any technology base outside of G. If G is not reachable, then the attempt to reach it will reveal what the boundary of this region is and may reveal why it cannot be reached.

The cellular automata systems that have been used for investigating SRPCs effectively permit the creation of new component parts out of empty space and the transformation of existing parts into empty space. This behaviour is not physically realistic, so one way of moving away from the abstract end of the physical realism axis of Fig. 1 is to consider systems in which parts cannot be created or destroyed. If parts cannot be created or destroyed then in order for a system to carry out any kind of construction operation at all, parts must be mobile. A structure made from mobile parts may fall apart if it is moved unless the environment also supports a means of connectivity by which connected parts can be made to move together.

Simulation environments of this type are termed 'kinematic environments' because they model movement and connectivity, but do not attempt to model any detailed physical basis for movement or connectivity. Systems embedded in such kinematic environments are called 'kinematic automata'. (A term first used by Burks[26]). Von Neumann's idea for a kinematic model of a self-replicating system, described in Section 2.10, inspired much of the work in this paper.

In Section 3, a kinematic simulation environment is presented which moves part of the way along the physical realism axis of Fig. 1 away from the abstract end and towards the physically realistic end. The main reason why we move only part of the way along the axis, rather than attempt a much more physically realistic simulation is that the computational cost of simulating a system increases as it becomes more physically realistic. The SRPC described in Section 4 takes about 10 days to simulate on a 2009-vintage desktop computer, a more physically realistic simulation could be expected to take longer than this.

3. A 3D Kinematic Environment with Six Part Types

This section describes a discrete space, discrete time 3D kinematic simulation environment called CBlocks3D. This is a development of the previously published 2D CBlocks environment,[32] having a greatly reduced set of component parts and not requiring any ability to create component parts out of empty space. There are six different types of part in the CBlocks3D environment: a signal propagation part (the *wire* part), a signal processing part (the *nor* part), a part for moving other parts (the *slide* part), a part for rotating other parts (the *rotate* part), a part for connecting other parts (the *fuse* part) and a part for disconnecting other parts (the *unfuse* part). These part types are chosen so as to be as simple as possible whilst spanning the operations required for construction within the environment. Boolean signals can pass between neighbouring parts. It takes one time unit for a part to respond to an input signal. In a single time unit, a part may move one unit in any one of six directions under the action of a *slide* part. When a part moves, all parts directly or indirectly connected to it also move.

3.1. Describing parts

We define six direction vectors

$$\begin{aligned} EAST &= (1,0,0) & WEST &= (-1,0,0), \\ NORTH &= (0,1,0) & SOUTH &= (0,-1,0), \\ FRONT &= (0,0,1) & BACK &= (0,0,-1), \end{aligned}$$

D denotes the set of these vectors

$$D = \{EAST,WEST,NORTH,SOUTH,FRONT,BACK\}.$$

We define the function

$$opposite((x,y,z)) = (-x,-y,-z),$$

L is the set $\{True,False\}$, and T denotes the set of part types,

$$T = \{wire,nor,slide,fuse,unfuse,rotate\}.$$

A part is completely described by the tuple

$$(P.location, P.primary, P.secondary, P.type, P.output, P.connect),$$

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

Table I. Part types in CBlocks3D.

1 Wire  *N,E,W,F,B:=S*	2 Nor  *N:=¬(E∨S∨W∨F∨B)*	3 Slide  If *S* then move the part that lies *NORTH* one unit *EAST*
4 Fuse  If *S* then connect the parts that lie *NORTH* and *NORTH-EAST*	5 Unfuse  If *S* then disconnect the parts that lie *NORTH* and *NORTH-EAST*	6 Rotate  If *S* then rotate the part that lies *NORTH* through 90 degrees

where

$$P.location \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$$
$$P.primary \in D$$
$$P.secondary \in D \backslash \{P.primary, opposite(P.primary)\}$$
$$P.type \in T$$
$$P.output \in L \times L \times L \times L \times L \times L$$
$$P.connect \in L \times L \times L \times L \times L \times L$$

P.location is a 3-tuple (x, y, z) which specifies the location of the part.

Two vectors are needed to specify the orientation of a part *P* in 3D discrete space. The primary axis *P.primary* is a vector that lies on the line from the centre of *P* to the centre of one face of *P* (this face is referred to as the *active face* of *P*). The secondary axis *P.secondary* is perpendicular to the primary axis. For example, in Table I the primary axis of each part points up the page (*NORTH*) and the secondary axis points to the right of the page (*EAST*).

The notation *X*[*Y*] is used to refer to the *Y*th element of the tuple *X*. It is convenient to use the direction vectors *D* to index the *P.output* and *P.connect* tuples, so we define that the vectors *NORTH*, *EAST*, *SOUTH*, *WEST*, *FRONT* and *BACK* can be used to index the 1st, 2nd, 3rd, 4th, 5th and 6th elements of a tuple, respectively.

$P.output[d] \in L$where $d \in D$ are the outputs of *P*. So for example if we have an isolated *nor* part *P* with $P.primary = EAST$, the values of its outputs will be $P.output[EAST] = True$ and $P.output[d] = False$ for all other $d \in D$.

$P.connect[d] \in L$ where $d \in D$ specify the connectivity state of *P*. If a part *P* is connected in a particular direction *d* to a neighbouring part *Q* then $P.connect[d] = True$ and also $Q.connect[opposite(d)] = True$. If a part *P* is not connected to its neighbour *Q* which lies in direction *d*, then $P.connect[d] = Q.connect[opposite(d)] = False$. If a part *P* has no neighbour in direction *d* then $P.connect[d] = False$. When a part moves, all parts connected to that part also move.

The faces of parts can be regarded as terminals through which Boolean signals can be passed between neighbouring parts. Neighbouring parts do not need to be connected in order for signals to pass between them. Each terminal of a part acts either as an input or as an output. If a terminal has no explicit definition, it is effectively an output producing a *False* signal. It takes one time unit for a signal to propagate from a part's inputs to its outputs or for a part to respond to signals at its inputs.

Table I describes the function of each type of part. A graphical representation and a colour coding scheme are shown for each part type. In Table I, the letters *N*, *E*, *S*, *W*, *F* and *B* are used to refer to the value of signals at the *NORTH*, *EAST*, *SOUTH*, *WEST*, *FRONT* and *BACK* faces of a part. In these diagrams, *NORTH* is up the page, *EAST* is to the right of the page and *FRONT* is out of the page. The Boolean ¬ (negation) and ∨ (OR) operators are used in Table I.

The *wire*, *nor* and *rotate* parts have rotational symmetry about their primary axis and can therefore be in any one of six functionally distinct orientations. The *slide*, *fuse* and *unfuse* parts have no rotational symmetry and can therefore be in any one of 24 distinct orientations.

Graphical representations for each part type in each possible orientation are shown in Table II and will be used later on for diagrams of systems in CBlocks3D. The letters beneath each graphic show the orientation of the part. The first letter gives the orientation of the primary axis and the second letter gives the orientation of the secondary axis. For example, a part in the orientation *SF* has its primary axis pointing in the *SOUTH* direction and its secondary axis pointing in the *FRONT* direction. For parts that are symmetrical about their primary axis the second letter is omitted.

Note that for the representations given in Table II *NORTH* is up the page and *FRONT* is out of the page. In later diagrams in this paper, the correspondence between page axes and CBlocks3D directions will vary because it is often easier to describe a mechanism viewed from one direction than from another. In these situations, the representation used for a part in a particular orientation will change accordingly. For example, a slide part in the *SF* orientation when *NORTH* is up the page and *FRONT* is out of the page will be represented in the same way as a slide part in the *EN* orientation when *WEST* is up the page and *NORTH* is out of the page.

3.2. *Evolution of a system of parts*

The following informal algorithm describes how a system of parts in the CBlocks3D environment evolves from one time unit to the next.

Step 1: Make a record of which *slide*, *rotate*, *fuse* and *unfuse* parts have *True* inputs. Calculate new outputs for every *nor* and *wire* part.

Step 2: Rotate any parts to be rotated by the action of any *rotate* parts. Alter any connections to be altered by the action of any *fuse* or *unfuse* parts.

Step 3: Work out whether any parts will be moved by the action of any *slide* parts.

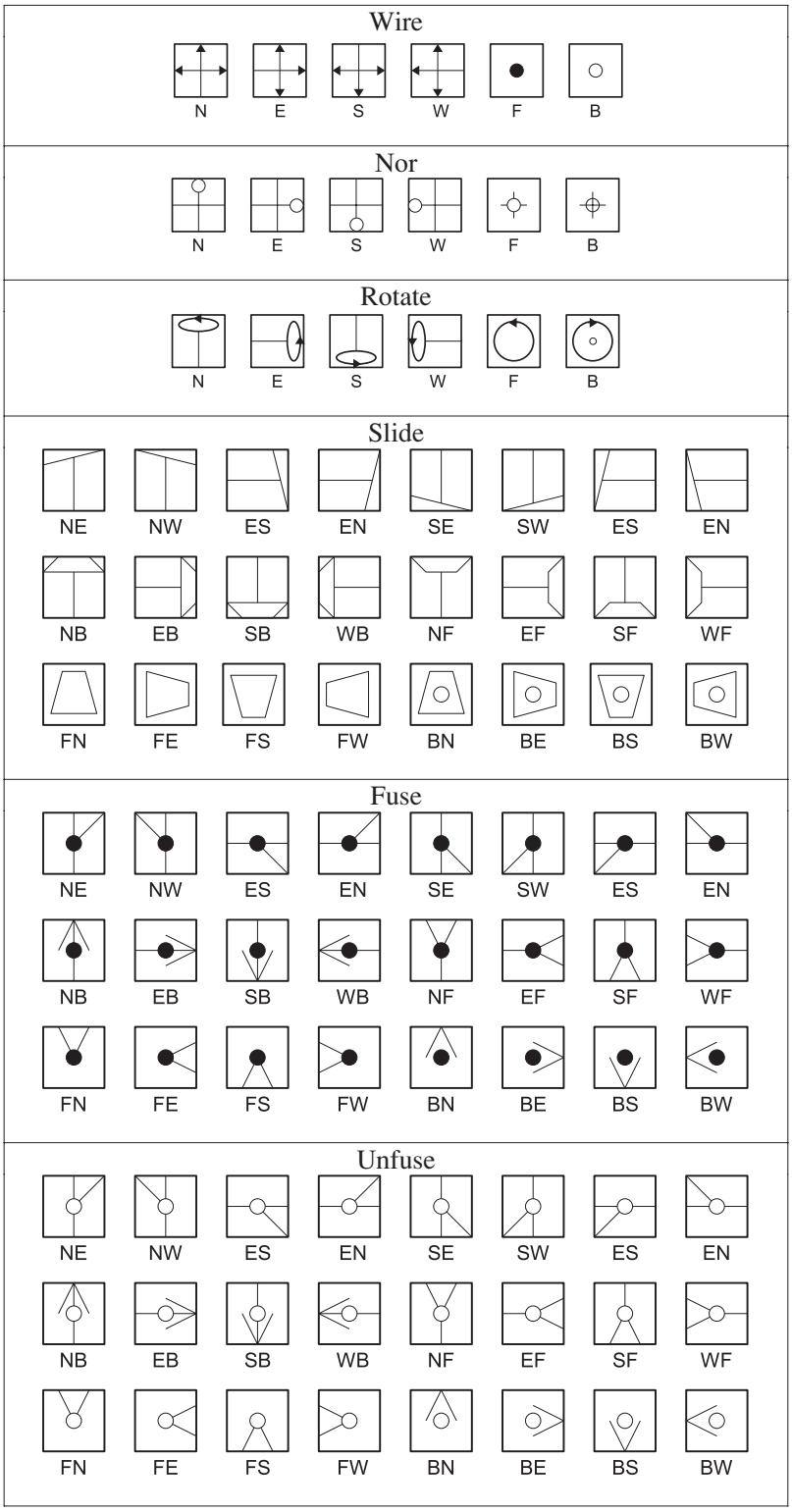
Step 4: Update the location of every part that is to be moved.

In step 3 when a slide part *S* moves another part *P* in direction *d*, all parts directly or indirectly connected to *P* will also move in direction *d*. Any parts that are pushed by this moving collection of parts will also move in direction *d*.

Conditions which result in an attempt to move any part in two different directions simultaneously are illegal. It is also illegal to attempt to move more than one part into an empty location, or to rotate a part about two different axes, or to both connect and disconnect a pair of parts at the same time.

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

Table II. Graphical representations of parts.



These 'illegal' conditions are specified to avoid having to deal with ambiguous situations that would otherwise arise, for example, in a collision between two structures moving in different directions. Rules could be written to resolve such situations, but since such situations do not arise in any of the mechanisms described in this paper, and since a description of these resolution rules would be lengthy,[33] we simply declare these situations illegal and rule that no mechanism should cause these conditions to occur.

An efficient implementation of this algorithm that draws upon some of the techniques used by the *Hashlife* algorithm[34] is available in the supplementary material for this paper.

3.3. Logic gates and circuits

Figure 2 shows how the *nor* part can be used as the basis for other logic gates.

A loop of wire parts can be used to store a sequence of signals. The signals will propagate around the loop endlessly.

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

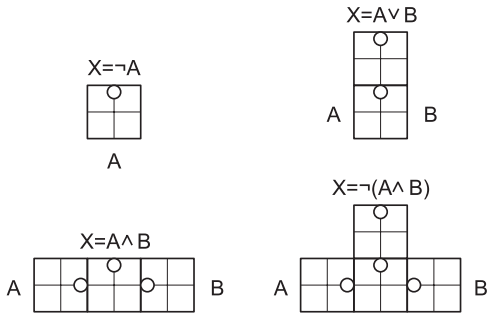


Fig. 2. NOT, OR, AND and NAND gates made from *nor* parts.

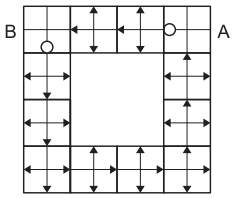


Fig. 3. A gated signal loop incorporating two *nor* parts.

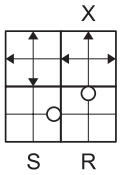


Fig. 4. A Set-Reset flip-flop made from *wire* and *nor* parts.

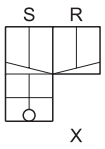


Fig. 5. A Set-Reset flip-flop that makes use of *slide* parts.

The addition of two *nor* parts to a signal loop (Fig. 3) allows a sequence of signals to be injected into an otherwise empty loop at input *A*, and also allows a loop to be reset into an empty state by applying a *True* signal at *B* for the length of time that it takes signals to propagate once around the loop. This type of signal loop is called a gated signal loop.

3.4. Flip-flops

Figures 4 and 5 show two different designs for Set-Reset flip-flops, each of which has different properties that determine its suitability in a given context. Each flip-flop has a set input *S*, a reset input *R* and an output *X*.

For the flip-flop shown in Fig. 4, a signal must be present on an input for a minimum of 4 time units in order for the *X* output of the flip-flop to change state.

The flip-flop shown in Fig. 5 is able to respond to a signal 1 time unit long by making use of slide parts to move the flip-flop's internal mechanism. The *nor* part in this flip-flop is not connected to either of the *slide* parts.

3.5. 1:4 pulse converter

The mechanism shown in Fig. 6 converts a *True* signal arriving at input *A* into a sequence of 4 *True* signals at output

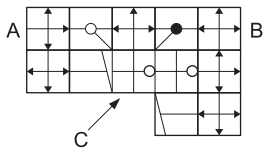


Fig. 6. 1:4 pulse converter.

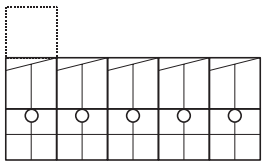


Fig. 7. A path along which parts can travel.

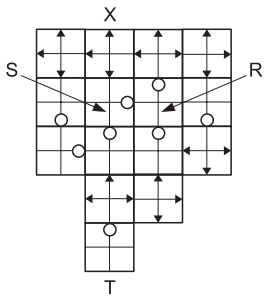


Fig. 8. A toggle flip-flop.

B. The mechanism makes use of a *slide* part to respond to a signal 1 time unit long. The mobile *nor* part *C* in Fig. 6 is normally connected to the rest of the mechanism. It is disconnected when a *True* signal arrives, and then reconnected again when the mechanism has finished its operation.

3.6. Transporting parts

Figure 7 shows how *slide* parts can be used to make a path along which other parts can travel. Each of the *slide* parts in Fig. 7 has a *nor* part providing it with a *True* signal. The dashed square stands for any part.

If any of the *nor* parts in Fig. 7 is fed a *True* signal on any of its inputs it will stop producing an output signal and cause any part travelling along the path to stop at the corresponding location on the path. This behaviour can be used as the basis for a mechanism for transporting parts to a specific location.

3.7. Counters and registers

Figure 8 shows how 11 parts can be added to the flip-flop of Fig. 4 to turn it into a toggle flip-flop which can be used as the basis for an asynchronous counter. A sequence of 4 *True* signals at the *T* input will cause the flip-flop to change state.

The faces that were used as *Set* and *Reset* inputs in Fig. 4 are no longer available in Fig. 8. However, because we are working in three dimensions different faces of the same parts can still serve as *Set* and *Reset* inputs (labelled *S* and *R* in Fig. 8).

A convention that will be used for diagrams of 3D structures later on in this paper is introduced in Fig. 9. All previous diagrams have been of planar circuits. Figure 9 shows a 3D circuit. The circuit can be thought of as being a

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

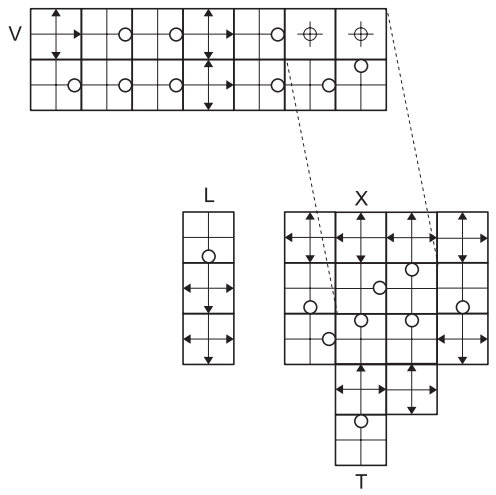


Fig. 9. A toggle flip-flop with Value (*V*) and Load (*L*) inputs.

stack of layers. The dashed lines connecting a point on the layer at the top of the figure with the layer beneath it shows how the layers are aligned with each other. The *FRONT*-most layer is drawn at the top of the page, with successively lower layers being 1 unit further in the *BACK* direction.

Figure 9 shows a toggle flip-flop with additional logic added to derive the *Set* and *Reset* signals from a *Load* signal (*L*) and a *Value* signal (*V*). A sequence of 4 *True* signals at *L* will cause the value present on *V* to be loaded into the flip-flop.

By having *N* toggle flip-flops and connecting a negative edge detector between the output of one flip-flop and the *T* input of the next, we can make an *N*-bit asynchronous counter that will increment in response to a sequence of 4 *True* signals on the *T* input of the first flip-flop.

3.8. Describing structures, visualisation and debugging

The supplementary material for this paper contains software for describing and visualising structures, and to help with the process of debugging complex structures.

To describe complex 3D structures made from connected substructures, a suite of C++ classes is used. In the same way that the part-level diagrams used in this section and Section 4 are drawn layer by layer, these classes allow small structures to be described layer by layer. Small structures can then be combined to form more complex structures.

Visualisation of 3D structures is invaluable for helping to design large complex structures. A utility was written using the OpenGL graphics library to render structures and to help with debugging. This utility allows users to navigate around a structure and view it from any location. Values of inputs to a structure can be set and a structure can be simulated one step at a time. Outputs from a structure can be sampled and written to a file.

4. An SRPC in the CBlocks3D Environment

Figure 10 shows a top-level schematic for an SRPC that has been devised in the CBlocks3D environment. In this schematic, lines between subsystems show which subsystems communicate with which other subsystems. Figure 11 shows a graphical representation of the SRPC after it has almost finished constructing a child machine. Several of the subsystems in the parent machine are labelled.

Characteristic subsystems are described in separate sections in the following part of this paper. Descriptions of most of the signal paths and logic that connect subsystems are omitted. The full specification of the system and a subtitled animation of the system operating are available in the supplementary material for this paper.

Three different types of diagram are used in the following sections. Schematic diagrams show mechanisms at a level of abstraction sufficient for understanding how a mechanism works without showing the individual component parts of the mechanism. Schematic diagrams are not given for simple mechanisms.

Part-level diagrams show how a mechanism is implemented. The representation used for parts is that given in Table II. Most mechanisms have a 3D structure and so use the convention for displaying multi-layered structures introduced in Fig. 9. Unless otherwise specified, all neighbouring parts in any structure are connected.

Although a 3D graphical representation projected onto a 2D page cannot show all of the parts within a mechanism, it is nevertheless useful to have a graphical representation. It will help the reader to recognise mechanisms when using the simulation and rendering software, and it will also assist with understanding the part-level diagrams. Graphical representations use the colour coding scheme introduced in Fig. 2. *Nor* and *wire* parts outputting *True* signals have a red square in the faces from which signals emerge. This can be seen clearly in Fig. 26.

Part-level diagrams and graphical representations contain an indicator showing how the figure is oriented with respect to the *NORTH*, *WEST* and *BACK* axes.

4.1. Detect mechanism

When the SRPC needs to replenish parts in its storage area it moves through its environment along a straight path from *EAST* to *WEST*. It has a single part-input orifice. Whenever the machine moves *WEST* by one unit a part from the environment may or may not have entered the orifice.

Figure 12 shows a mechanism that is able to detect when a part has entered the machine and Fig. 13 shows a graphical representation of the mechanism. This mechanism is called the *detect* mechanism, it has two signal inputs. One is the *part-request* input into which a *True* signal is sent when the *detect* mechanism should begin looking for a part. The other input is the *enable/disable* input. Whenever this input has a *True* signal applied to it, the *part-request* input has no effect and the activity of the *detect* mechanism is halted. When the *enable/disable* input transitions from *True* to *False*, it has the effect of both enabling the mechanism and also injecting a *True* signal into the *part request* input.

The operation of the *detect* mechanism is as follows. *Slide* part *A* causes the whole mechanism (and the whole machine, since the mechanism is connected to the rest of the machine) to move *WEST* one unit. *Unfuse* parts *B* and *C* disconnect structures *D* and *E* from the rest of the mechanism. *Slide* part *F* moves structure *D NORTH* by one unit. If there is a part at

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

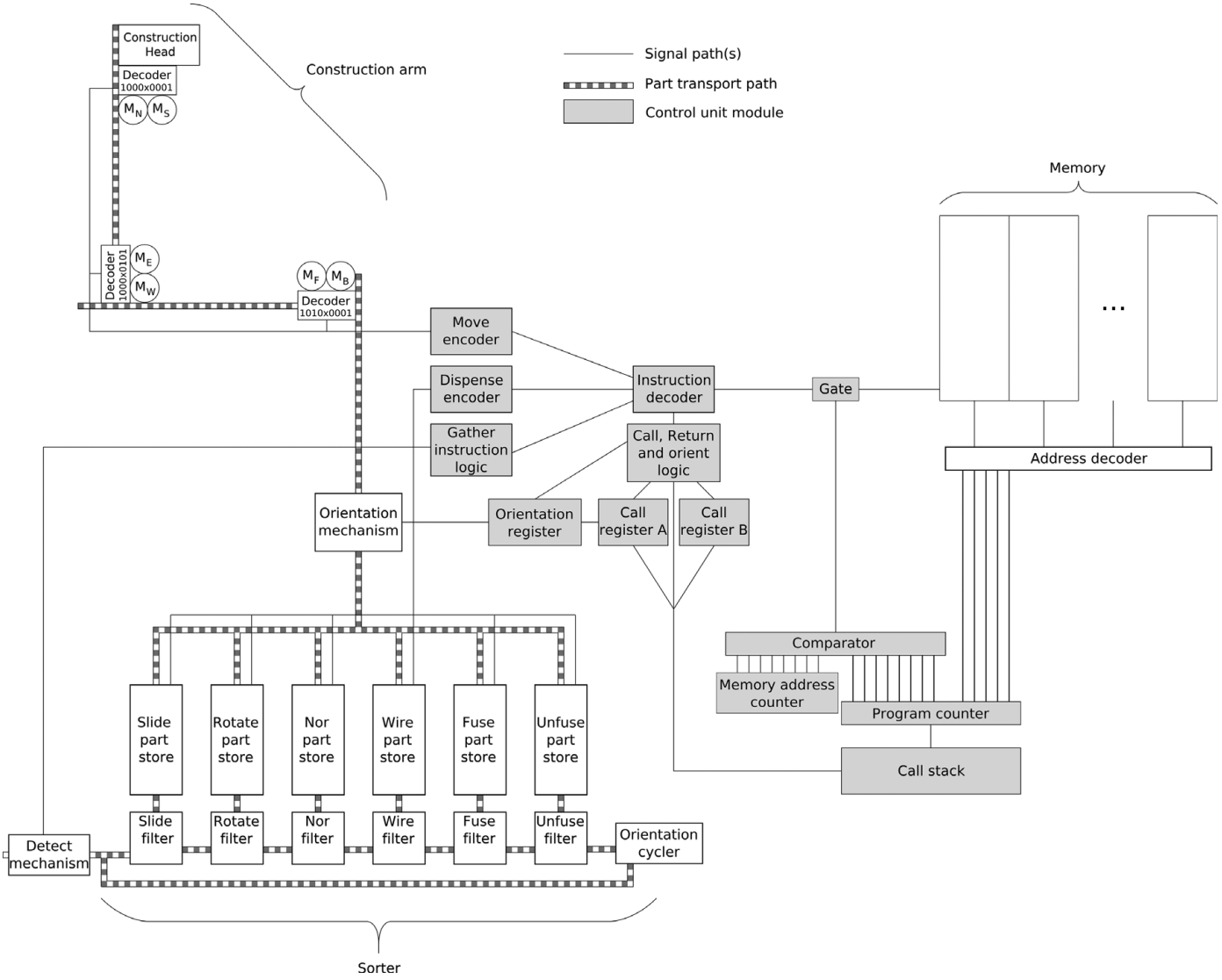


Fig. 10. Top-level schematic of the SRPC.

G then *D* will push this part which will in turn push structure *E NORTH* by one unit.

After *slide* parts *H* and *I* have moved *D* and *E* back to their original positions, *fuse* parts *J* and *K* will reconnect *D* and *E* to the rest of the mechanism. Only if *E* moved *NORTH* will a signal derived from part *L* emerge from the *part present* output. At the same time, the detected part will emerge at the *part output* location. If *D* moved *NORTH* but *E* did not then a signal derived from part *M* will travel to wire *N* and repeat the operation.

4.2. *Sorter mechanism*

A description of a mechanism for sorting parts in the CBlocks3D environment has already been published[35] and so is not described in full here. The operation of this mechanism can be summarised as follows: A part enters the *sorter* mechanism from the *part-output* location of the *detect* mechanism. The part is then fed in turn to mechanisms called *filters*. Each *filter* can recognise a particular type of part. The *slide*, *rotate*, *nor* and *wire filters* can recognise parts in any orientation in which the primary axis of a part points *SOUTH*. The *fuse* and *unfuse filters* can recognise parts in the *SF* orientation. When a *filter* recognises a part, the part is passed to a *storage* mechanism. If a part passes through all six filters without being recognised then its orientation is altered and it is passed back to the first *filter* for another attempt at recognition.

Figure 14 is a schematic diagram of the *sorter* mechanism which also shows how it is connected to the *detect* mechanism. Figure 15 shows a graphical representation of both mechanisms.

4.3. *Storage mechanisms*

There is a separate *storage* mechanism for each type of part. Each *storage* mechanism is based around a 2D rectangle of parts. Figure 16 shows a part-level diagram of the mechanism and Fig. 17 shows a graphical representation of the mechanism. Parts shaded in the same colour in Fig. 16 form connected structures. The connectivity between these structures is as follows: The *storage adder* mechanism is not connected to any other mechanism except the rectangle of parts, to which it is connected via the *WEST* face of the

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

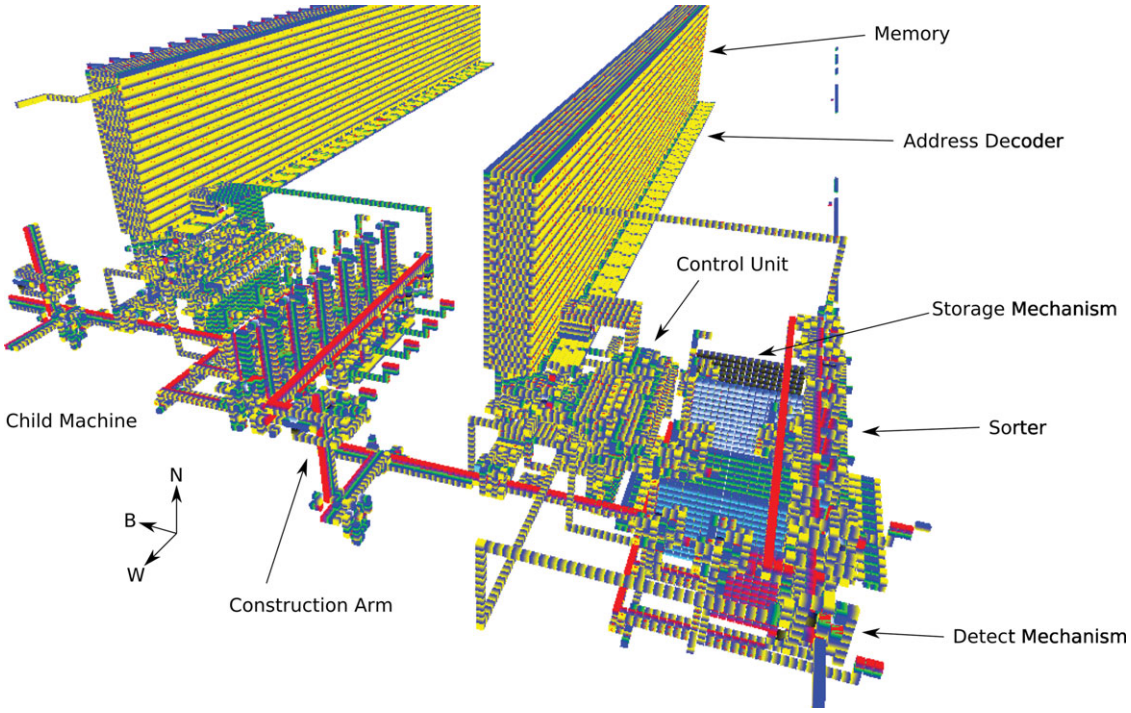


Fig. 11. Graphical representation of the SRPC.

wire part *SOUTH* of *unfuse* part *G*. The *storage dispenser* mechanism is not connected to any other mechanism except the rectangle of parts, to which it is connected via the *EAST* face of the southmost of *unfuse* parts W. The part path *AF* and all parts connected to it are not connected to any other structure. This structure is held in place by surrounding structures.

When a part is recognised by the *sorter*, it is passed on to a *storage* mechanism and added to the *FRONT* edge of the rectangle of parts. A part-level diagram of the mechanism for adding parts onto the rectangle is shown towards the bottom right hand corner of Fig. 16. This mechanism is called the *storage adder*.

A part *p* enters the *storage* mechanism at *A* at the same time that a signal enters at *B*. Part *C* in the *storage adder* prevents *p* from moving *SOUTH* past *slide* part *D*. *Slide* part *D* moves *p* into the next available empty location on the rectangle of parts and then *fuse* parts *E* connect *p* to the rectangle.

A signal derived from *B* enters the *storage adder* through part *F*. This causes *unfuse* part *G* to disconnect the *storage adder* from the rectangle. The adder then moves one unit *NORTH* and reconnects to the rectangle.

If the *storage adder* reaches its northmost location, then a signal from *H* enters the mechanism through part *I* and causes the *storage adder* to move 12 units *SOUTH*. Then a signal enters *J* causing the rectangle to be temporarily disconnected by *unfuse* part *K*, moved *BACK* 1 unit by *slide* part *L* and then reconnected by *fuse* part *M*. This creates space for a new column of parts.

If as a consequence of moving the rectangle *BACK* and creating space for a new column of parts the *storage* mechanism becomes full, then a signal from *nor* part *N* will enter part *O* and propagate to part *P*. With the *storage* adder in its *SOUTH* most location, a signal at *P* will cause the output of *C* to be *False* and will also cause *slide* part *D* to be inactive with the result that any part coming into the full *storage* mechanism from *A* will pass all the way *SOUTH* to *slide* part *Q*, which will then move the part out of the *storage mechanism* along path *R* and return it to the environment.

When the machine's control unit requests a part from a *storage* mechanism the part is retrieved from the *BACK* edge of the rectangle of parts. The mechanism for doing this is called the *storage dispenser* and includes a serial decoder which enables each *storage dispenser* to be activated by a different sequence of signals. A part-level diagram for the *storage dispenser* is shown towards the top right hand corner of Fig. 16.

A serially encoded signal enters the *storage* mechanism at *S* and enters the *storage dispenser* through *wire* part T. *U* is a decoder that decodes the serially encoded signal and causes the *storage dispenser* to move *NORTH* by activating *slide* part *V*. *Unfuse* parts *W* then disconnect a part from the rectangle of parts and *slide* part *X* causes this part to exit the mechanism along part-path *AF*.

If the *storage dispenser* reaches its northmost position, then the column from which it has been dispensing parts is empty. A signal from *Y* enters *Z* which results in *slide* part *AA* sliding the *storage dispenser* 12 units *SOUTH*. As the *storage dispenser* moves *SOUTH*, a signal from *AB* enters signal path *AC* and is passed onto *AD*, which then causes *AE* to move the *storage dispenser* 1 unit in the *FRONT* direction, ready to access the next column of parts.

Table III shows the serial encoding used for dispensing each type of part. In this table, the rightmost bit is the leading bit as the sequence propagates along a path.

The maximum size of a *storage* mechanism rectangle is $12 \times 17 = 204$ parts. There is a limit on how close the *storage adder* and *storage dispenser* can be can be without interfering

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

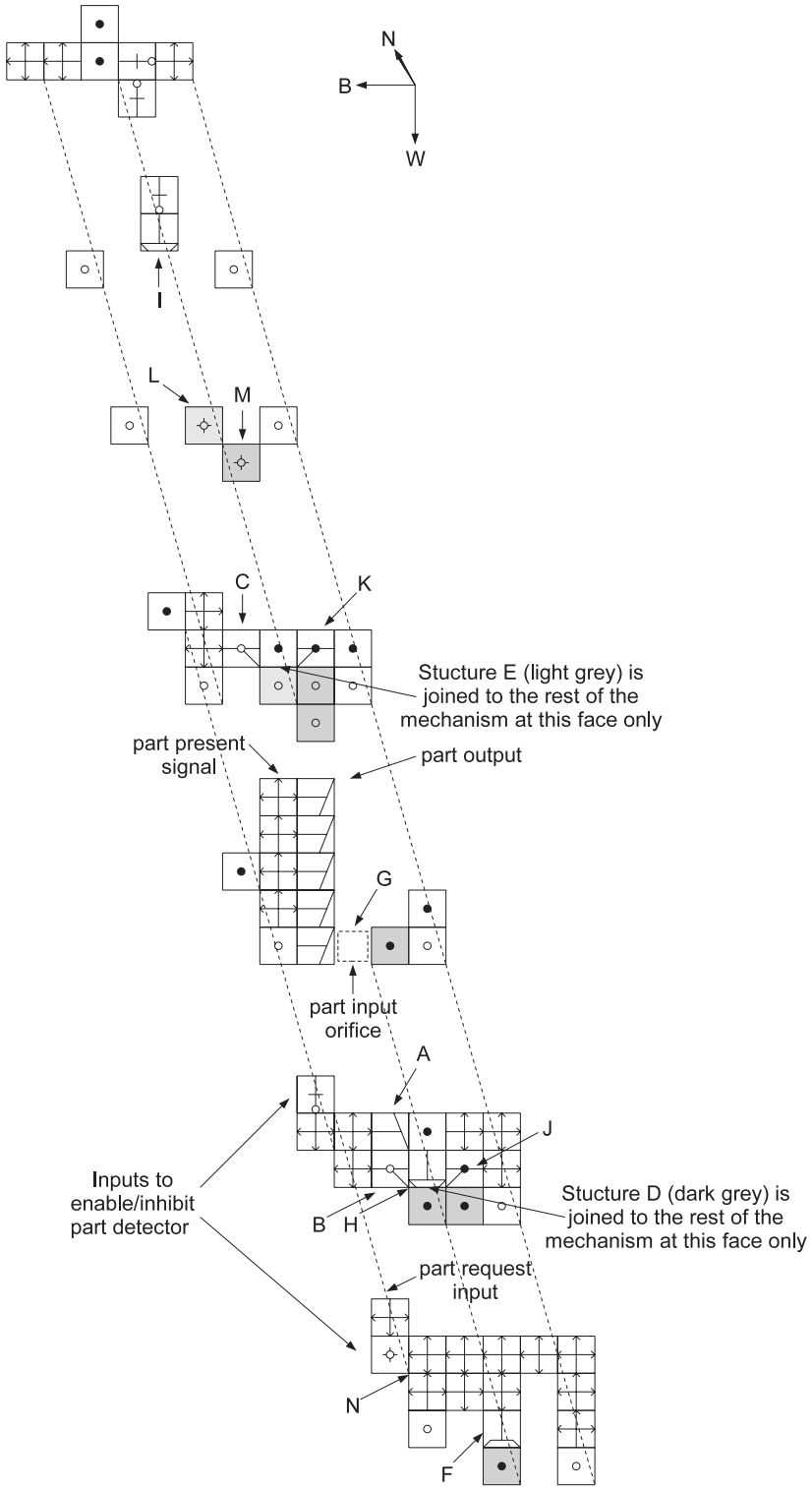


Fig. 12. The *detect* mechanism.

with each other's operation. For this reason, the effective capacity of a *storage* mechanism is 180 parts.

4.4. Orientation mechanism

The *orientation* mechanism takes as its input a part in the *SF* orientation and then rotates the part into a specified orientation. Since there are 24 possible orientations, a 5 bit word is required to specify an orientation.

Let us assume we have a 5 bit word stored in a register. The problem of designing the *orientation* mechanism then becomes: how can these 5 bits be fed to a mechanism operating on a target part in such a way that different words will result in different orientations of the target part and that there is at least one word resulting in each possible orientation.

One simple mechanism of this type is a part path running, say, from *FRONT* to *BACK* with each bit from the register

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

Table III. Serial encoding used for dispensing parts.

Part type	Encoding
Slide	100000001
Rotate	100010001
Nor	101010001
Wire	101000001
Fuse	100000101
UnFuse	100010101

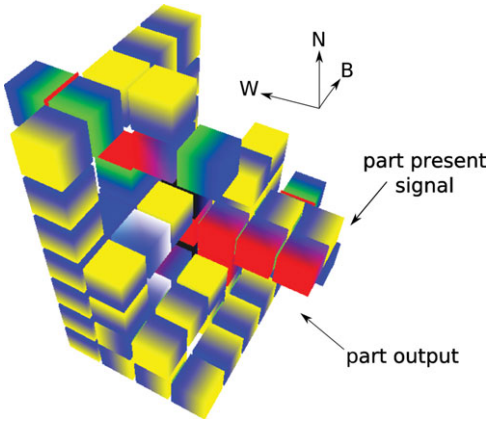


Fig. 13. A graphical representation of the *detect* mechanism.

leading to a *rotate* part that will operate on parts passing along this path. Clearly, some of the *rotate* parts will need to rotate about different axes from others in order to span the whole range of orientations. Figure 18 shows a mechanism that was found to exhibit the desired behaviour. For each possible value of the orientation register, Table IV shows what the resulting orientation is for parts emerging from the *part-output* location of the *orientation* mechanism.

4.5. Construction arm

Section 3.6 describes how a path for transporting parts can be made in such a way that a part can be stopped at any location along the path. By using three such paths set orthogonally and joined so that each path can slide back and forth along one axis, we can make a mechanism that can move a part to a particular location in a cubic region. We call this mechanism the *construction arm*.

At each joint in the *construction arm*, decoders are used to decode any sequence of signals fed into a signal path joined to and running along with the part paths. The outputs of these decoders lead to *slide* parts which cause each of the different segments of the *construction arm* to move back and forth along one axis. Table V shows the encoding used for each movement of the *construction arm*.

Figure 19 shows a single joint between two signal paths in the *construction arm*. Figure 20 shows a graphical representation of the entire *construction arm*.

The operation of the joint shown in Fig. 19 is as follows. *Nor* part *A* prevents parts from moving further along the path than the position of the joint so that parts can be passed onto the next path at right angles to this one.

Signals from the signal path that runs along with the part path enter the joint at *B*. Output *C* is used to pass this signal onto the next signal path. Starting at part *D* is a decoder that is used to decode two sequences that differ by 1 bit (see Table V). If either sequence is decoded, then the joint is disconnected from the part path by *unfuse* part *E*. Depending on which of the two sequences was decoded one of the *slide* parts *F* causes the joint to move in one direction or the other. After this *fuse* part *G* reconnects the joint to the part path.

At the very end of the *construction arm* is a mechanism called the *construction head*. This contains decoders and *slide* parts that fulfill the same function as in a joint. The *construction head* contains *fuse* parts to connect newly placed parts to the structure being constructed. The *construction*

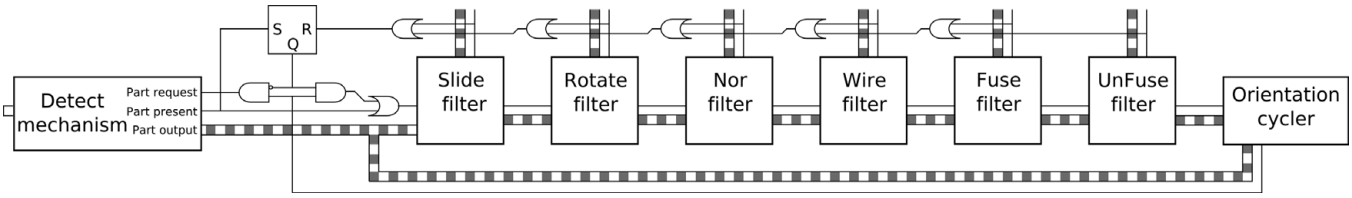


Fig. 14. A schematic diagram of the *sorter* mechanism connected to the *detect* mechanism.

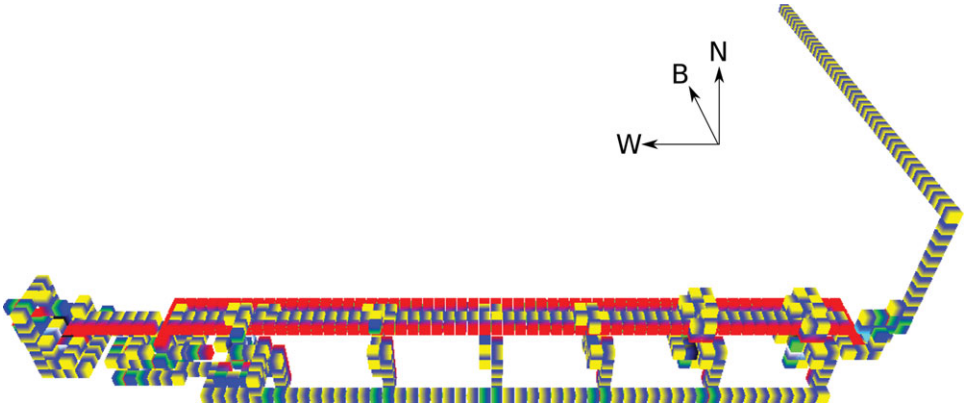


Fig. 15. A graphical representation of the *sorter* mechanism connected to the *detect* mechanism.

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

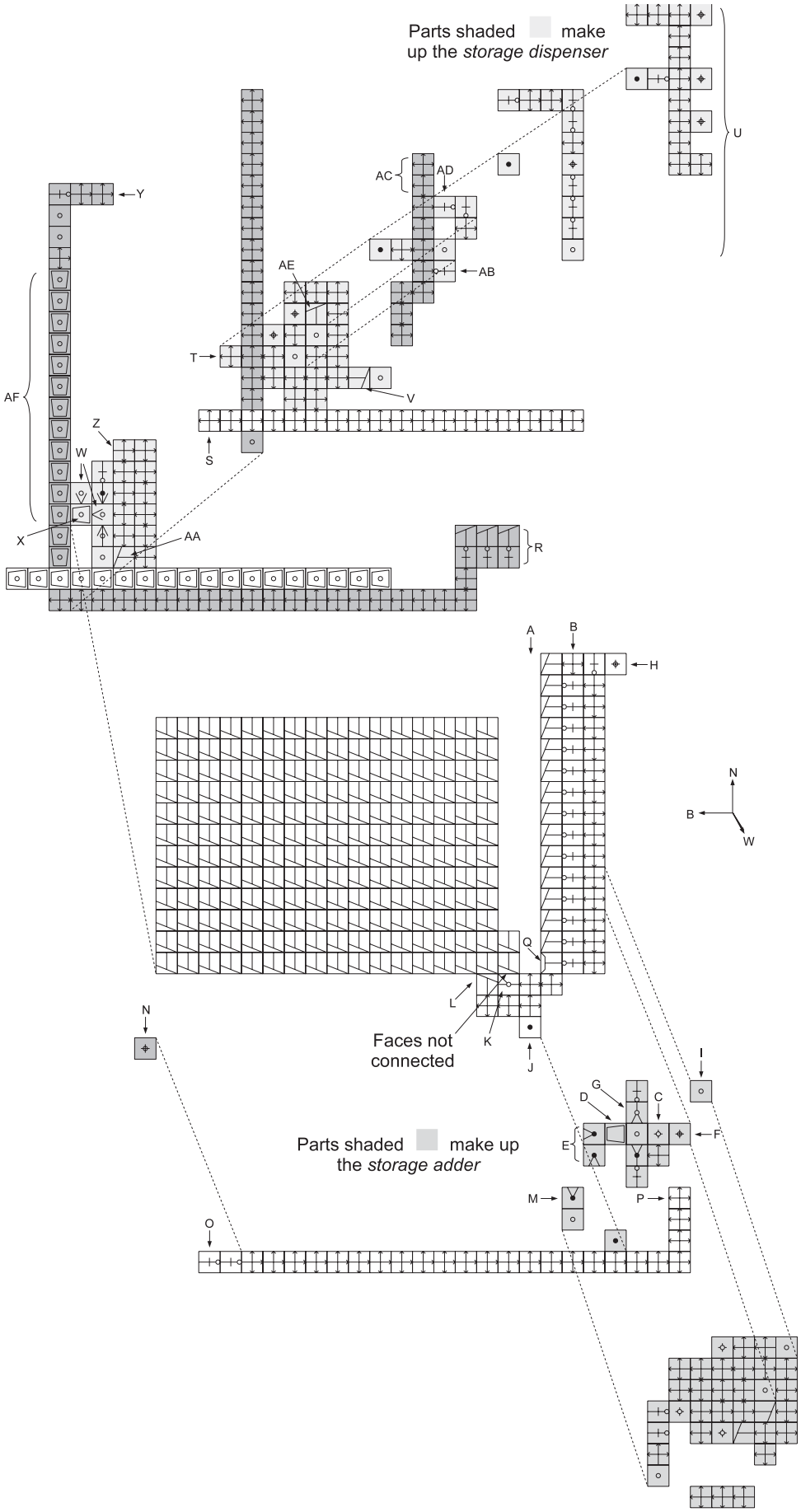


Fig. 16. The *storage mechanism* for *slide* parts.

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

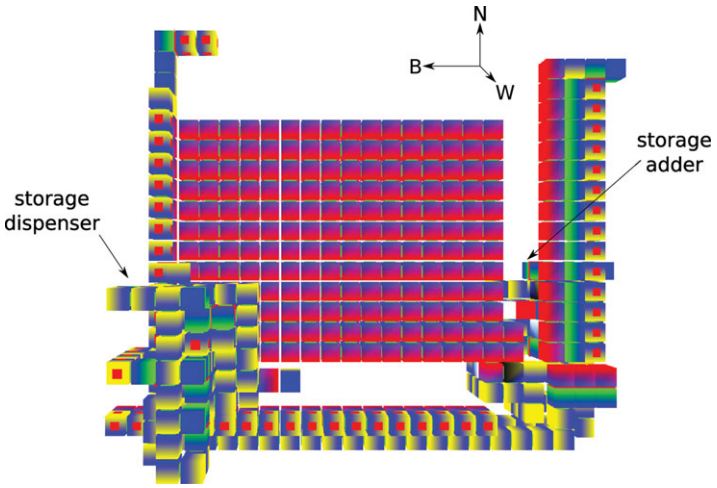


Fig. 17. A graphical representation of the *storage mechanism*.

head also contains mechanisms that can be used to attach and detach the *construction head* to and from the construction. One attachment/detachment mechanism lies on the *EAST* side of the *construction head* and another lies on the *WEST* side. These are used to make sure that when the machine is collecting parts to fill the *storage* mechanisms, the construction is not left behind. They are also used occasionally during construction to pull the construction in a particular direction in order to operate on a different part of the construction.

The signals used to activate the attachment and detachment mechanisms are taken from the same decoder that is used to decode *NORTH* and *SOUTH* movements of the *construction head* respectively. This means that in order to be attached to a construction the *construction head* must be positioned so that either of the attachment/detachment mechanisms is against the construction and then the *construction head* must be moved *NORTH*. Immediately after moving *NORTH* the *construction head* will be attached to the construction. To detach the *construction head* from the construction it must be moved *SOUTH*. Immediately after moving *SOUTH* (the construction will also move *SOUTH* because it is attached) the *construction head* will be detached from the construction.

4.6. Memory

Having described the *construction arm*, *part dispenser* and *orientation* mechanism, it is now possible to describe the instruction set that is used to program the SRPC and the memory that is used to store these instructions. The

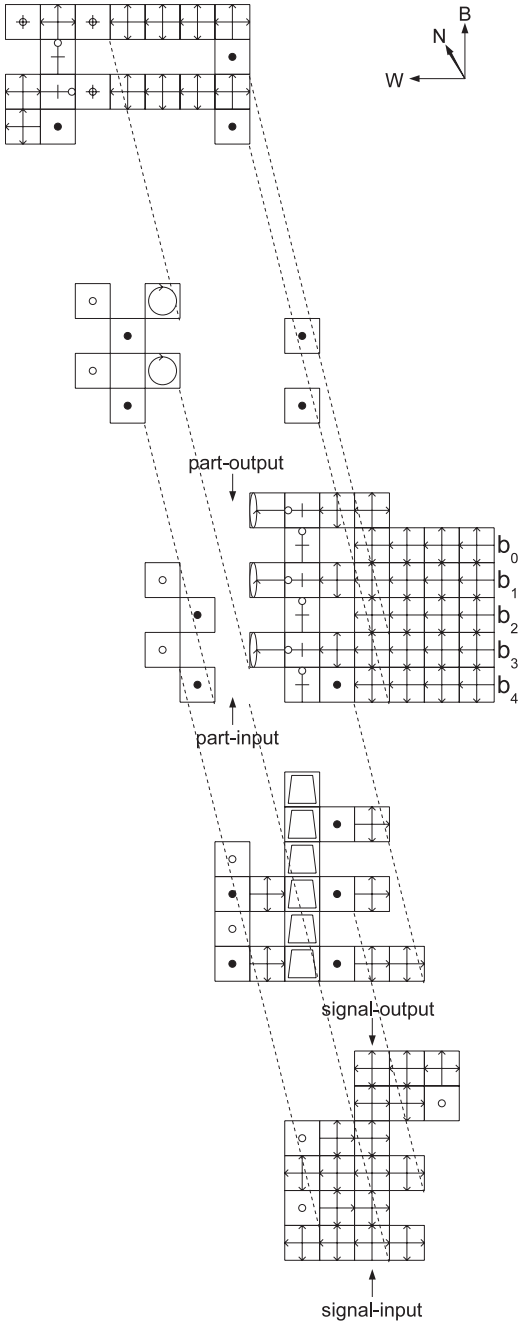


Fig. 18. The *orientation* mechanism.

Table IV. Orientation of parts at *part-output* for different values of the orientation register.

Value	Orientation	Value	Orientation	Value	Orientation	Value	Orientation
00,000	EB	01,000	WF	10,000	EF	11,000	WB
00,001	EN	01,001	WS	10,001	ES	11,001	WN
00,010	BW	01,010	FE	10,010	BE	11,010	FW
00,011	NW	01,011	SE	10,011	NE	11,011	SW
00,100	SB	01,100	NF	10,100	SF	11,100	NB
00,101	BN	01,101	FS	10,101	BS	11,101	FN
00,110	BN	01,110	FS	10,110	BS	11,110	FN
00,111	NF	01,111	SB	10,111	NB	11,111	SF

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

Table V. Serial encoding used for *construction arm* movements.

Direction	Encoding
NORTH	100000001
SOUTH	100010001
EAST	101010001
WEST	101000001
BACK	100000101
FRONT	100010101

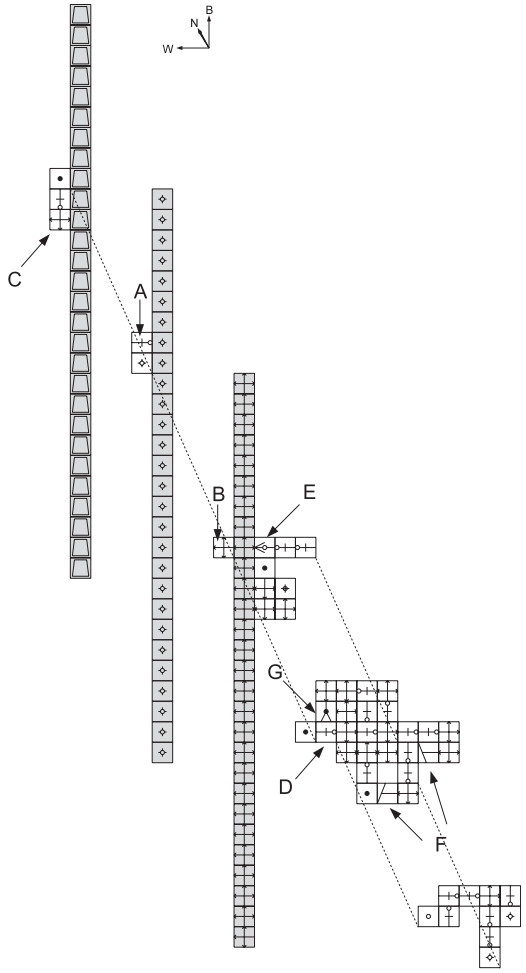


Fig. 19. A single joint between two paths in the *construction arm*.

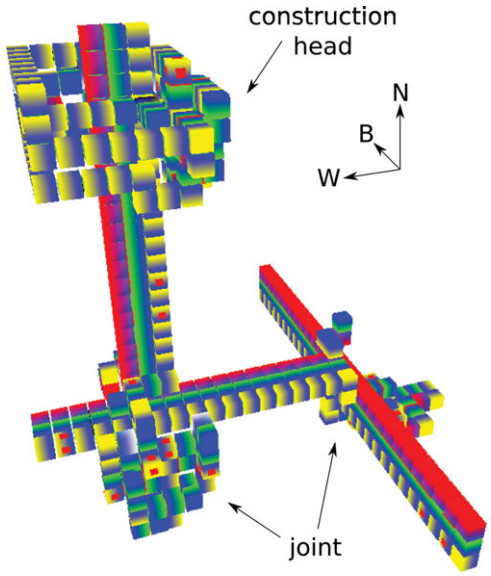


Fig. 20. A graphical representation of the *construction arm*.

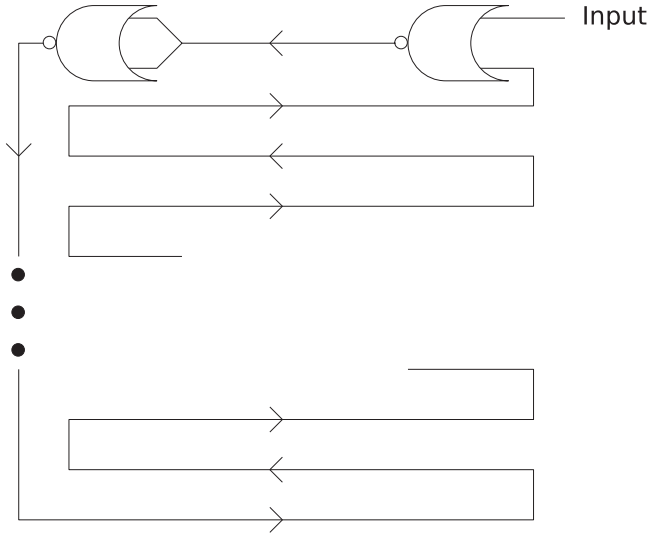


Fig. 21. The arrangement of a memory signal loop.

arrangement of the memory is described in this section and the instruction set is described in Section 4.8.

There are a number of different ways in which information storage structures may be implemented using the part types available. We have already seen in Section 3.3 that a pattern of signals will propagate endlessly around a signal loop, which has some similarity to the delay line stores that were used in early computer memories. If we define the storage density of a memory as the number of bits of information that can be stored per unit volume, then, not including circuitry required to extract signals from such a memory, the density of this type of memory is 1 bit per unit volume.

An upper limit for the density of a maximal-density storage structure can be given by considering the number of bits that it takes to describe the state of a single cubic volume of space, taking into account the part type contained within a volume, its orientation and its connectivity state. Clearly, the density of this type of memory will be higher than the density of a memory based on signal loops. However, the mechanisms involved in reading, writing and copying this type of memory are not nearly so straightforward as those for a memory based on signal loops. We choose to use gated signal loops as described in Section 3.3 because it simplifies the problem of copying the contents of one memory unit to another when the contents of the parent SRPC's memory need to be copied into the child machine. The memory units simply need to be held in contact for the length of time that it takes a signal to propagate around the loop.

The signal loop used as the basis for the SRPC's memory is a 256-part loop that is folded up so as to fit in a rectangular area 8 units *FRONT* to *BACK* and 32 units *SOUTH* to *NORTH*. Figure 21 shows how each memory loop is arranged.

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

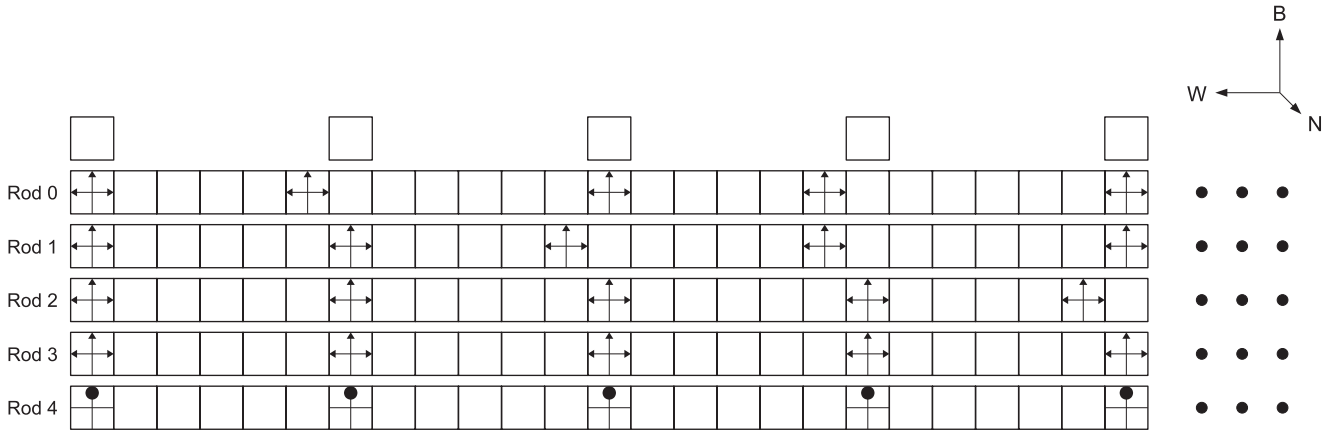


Fig. 22. The structure of the address decoder.

Each 256-word memory module is made using six gated signal loops placed side-by-side from *WEST* to *EAST*. The SRPC contains 32 memory modules giving a total of 8,192 6-bit words. Each module has a 6-bit wide AND gate connected to its output. The outputs from the AND gates from each module are ORed together to produce a single output for the whole memory unit. By activating one 6-bit AND gate the contents of a single memory module can be made to appear on the output of the memory unit.

Within a 6-bit memory module the output from each memory loop is offset 1 bit FRONT-ward from that of the previous loop. Also each memory module is six units further along the ORing mechanism than the previous module. The contents of the memory therefore have to be arranged in such a way as to compensate for these differences in offset so that the correct word is available at the output at the correct time. Let us take the signal loop that produces output bit 0 of the first memory module as a reference loop. Then the contents of signal loop b of module m have to be offset by $b+6m$ units with respect to the reference loop.

4.7. Address decoder

The memory consists of 32 modules each containing 256 words. The output of each memory module passes through a 6-bit AND gate which can be used to select which of the 32 modules is active. The outputs of these AND gates are then ORed together so that there is a single output for the whole memory unit.

The *address decoder* has five single-bit inputs. Based on the value of these inputs it produces a *True* signal on one of 32 outputs and a *False* signal on the other 31 outputs. The *address decoder* consists of five rods running parallel to each other and to the memory output AND gates. Each rod corresponds to an input bit of the *address decoder*. Each rod can be shifted in position one unit eastward. Figure 22 shows part of each rod. Parts that play no role in the operation of the address decoder are shown as blank squares to avoid clutter. Each rod is made from *NORTH* facing *wire* parts except for 32 special parts, called path-forming parts, that have their primary axis in the *BACK* direction. The spacing of these path-forming parts is such that for each of the 32 possible combinations of positions for the five rods, exactly one signal path is formed from the alignment of path-forming parts so

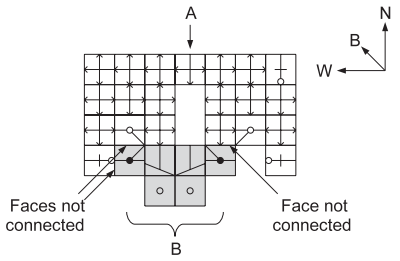


Fig. 23. The *selector* mechanism for converting a *True* or *False* value to an offset.

that a single memory module is selected. This is achieved by locating path-forming part n at a position $6n - p_r(n)$ from the *WEST* end of the rod, where $0 \leq n \leq 31$ and p is determined for each part of each rod r as follows: $p_r(n) = 1$ if $n \bmod 2^{r+1} > 2^r - 1$, $p_r(n) = 0$ otherwise

A mechanism is required to convert the output of the upper 5 bits of the *program counter* into an offset for each address decoder rod. Figure 23 shows a single bit of this mechanism, called the *selector* mechanism. If a logic 1 signal is input at *A* it will cause structure *B* (and anything connected to it) to adopt the EAST position. Otherwise *B* will adopt the WEST position. The *fuse* and *unfuse* parts in the *selector* mechanism ensure that when A is stable, structure B is connected to the rest of the *selector* mechanism.

4.8. Instruction set

The instruction set must contain instructions for moving the construction head in each of 6 directions, for specifying each of 24 orientations and for dispensing 6 types of part. A single 6-bit instruction word can be used to encode all of these operations.

An additional factor to account for in the design of the instruction set is that the SRPC requires some mechanism for repeating sequences of instructions. In order for the memory to be able to store the instruction sequence for replicating the entire machine, the instruction sequence needed to construct the memory must fit into the memory itself, with enough space left over for the remainder of the instruction sequence for replicating the entire machine. This is only possible if the memory has a regular structure that can be described by a repeated sequence of instructions. Enabling instruction

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

Table VI. Instruction encoding scheme.

Instruction	Encoding	Meaning
DEL	000101	Dispense a *wire* part
NOR	000010	Dispense a *nor* part
SLI	000111	Dispense a *slide* part
ROT	000100	Dispense a *rotate* part
FUS	000011	Dispense a *fuse* part
UFS	000110	Dispense a *unfuse* part
ORIENT p	1pppp	Set the orientation register to *p*
NORTH	001000	Move the construction arm *NORTH*
EAST	001101	Move the construction arm *EAST*
SOUTH	001100	Move the construction arm *SOUTH*
WEST	001001	Move the construction arm *WEST*
FRONT	001110	Move the construction arm *FRONT*
BACK	001010	Move the construction arm *BACK*
CALL a	1aaaaa 1aaaaa	Call the subroutine at address 4096+4*a*
RETURN	011000	Return from the current subroutine
GATHER	010000	Begin collecting parts.
NOP	000000	No operation – do nothing

sequences to be repeated also reduces the number of instructions needed for constructing other parts of the SRPC that also have duplication or regularity in their structure.

Two possible mechanisms for repeating sequences of instructions are loops and subroutines. Since a subroutine call mechanism can be used to emulate the effect of a loop (at the expense of a slightly increased number of instructions), and since a subroutine call mechanism can be used to remove several types of redundancy within a sequence of instructions, whereas a loop mechanism can only remove contiguous repeated sequences, it was decided to implement a subroutine call mechanism but not a loop mechanism.

Table VI shows the complete instruction set for the SRPC along with the encoding chosen for each instruction.

A CALL instruction requires two instruction words and contains a 10-bit operand *a*. The destination address for a call instruction is 4,096+4*a*. This means that all subroutines must be placed within the upper 4,096 words of the 8,192 word memory and must begin on a four word boundary. This results in some wastage of memory when the length of a subroutine is not a multiple of four words but this wastage is more than made up for by having the CALL instruction fit into two words rather than three.

Note that the CALL and ORIENT instructions both use an encoding in which the most significant bit (MSB) of the instruction word is 1. A sequence of two consecutive instruction words with MSB set to 1 will be interpreted as a CALL instruction, whereas a single instruction word with MSB set to 1 will be interpreted as an ORIENT instruction. The only restriction that this places on the instruction sequence is that an ORIENT instruction may not precede a CALL instruction. This situation can be prevented by inserting a NOP instruction whenever this situation occurs.

4.9. Control unit

The *control unit* is the part of the SRPC that deals with the flow of program execution. It keeps track of which instruction is to be executed next, manages changes in program flow (resulting from the execution of CALL and RETURN instructions) and is responsible for decoding and interpreting all instructions.

4.10. Program counter and call stack

Central to the operation of the *control unit* is the *program counter*. This is a 13-bit register storing the address in memory of the next instruction word to be fetched from memory and executed. The *program counter* increments by one every time a word is fetched from memory. Only the lower 12-bits of the *program counter* increment. After interpreting the instruction word at address 4,095, program flow continues at address 0 rather than address 4,096. The effect of this is that after it has finished constructing the child machine, the parent machine restarts execution of the construction program and begins constructing a second child machine.

When a CALL instruction is executed the *program counter* is pushed onto the *call stack* and the 10-bit operand of the CALL instruction is loaded into bits 2–11 of the *program counter*. Bits 0 and 1 of the *program counter* are set to 0 and bit 12 is set to 1. This has the effect of multiplying the operand of the CALL instruction by 4 and adding 4,096.

When a RETURN instruction is executed the *program counter* is popped from the *call stack* and then incremented by 1 so as to continue execution from the instruction after that, which led to the subroutine call.

The *call stack* is implemented as an array of flip-flops that can move back and forth *SOUTH* of the *program counter*. Before a value is pushed onto the *call stack*, the whole array is moved backwards. After popping a value from the *call stack*, the whole array is moved forward.

Figure 24 shows a single bit of the *program counter* and the interface between the *program counter* and the *call stack*. Figure 25 shows a graphical representation of the *program counter* and *call stack*.

4.11. Memory address counter

The word being output by the memory at a point in time depends upon which of the 32 memory modules is selected

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

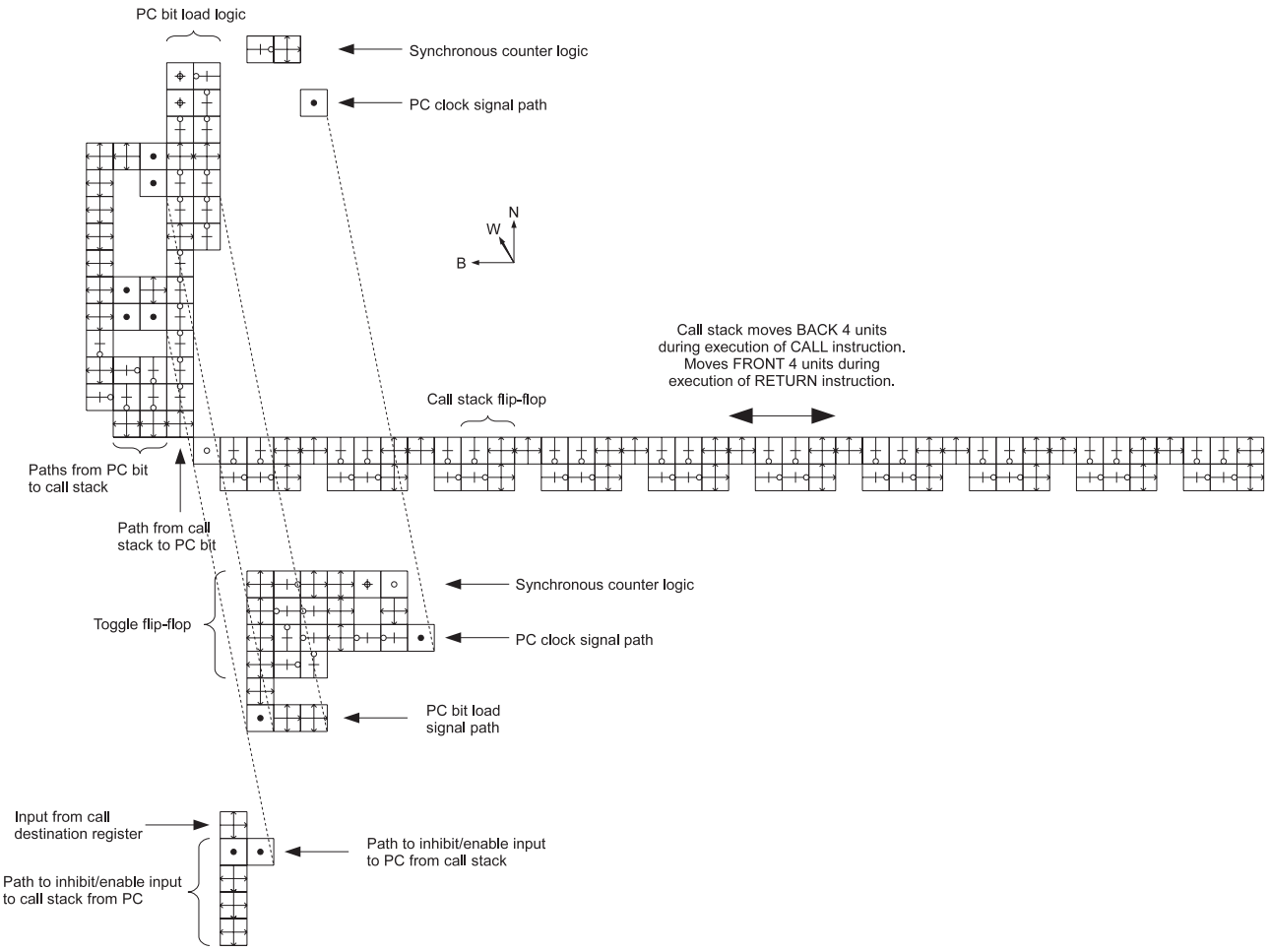


Fig. 24. A single bit of the *program counter* and *call stack*.

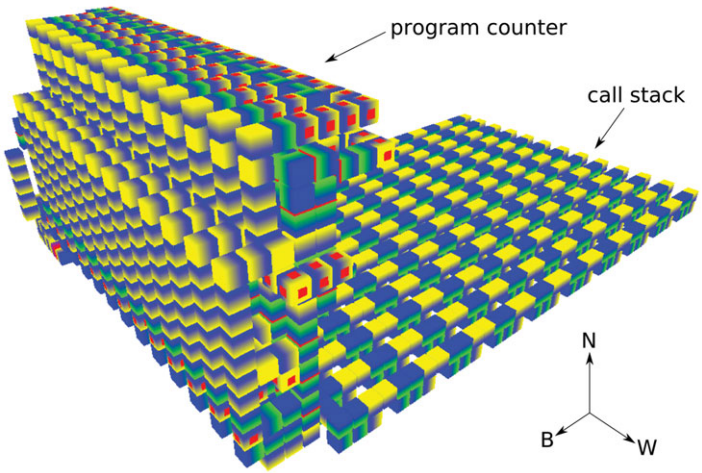


Fig. 25. A graphical representation of the *program counter* and *call stack*.

and the time t. If we define t_0 as the time at which memory module 0 outputs word 0 when module 0 is selected then the word being output at time t when memory module m is selected is given by $(t - t_0) \bmod 256 + 256\,m$.

The selected module m is determined by the upper 5 bits of the *program counter*. In order to obtain the memory word addressed by the *program counter*, we need to sample the output of the memory unit at a time when the address of the offset within a single memory module of the word being output matches the lower 8 bits of the *program counter*. In order to do this comparison, we can use an 8-bit counter that increments by 1 every time unit to keep count of which word is currently at the output of the memory. This counter is called the *memory address counter*, a graphical representation is shown in Fig. 26. Bits 0 to 4 of the *memory address counter* are made from signal loops containing patterns of signals that cause an incrementing binary sequence to appear on outputs 0 to 4. Bits 5 to 7 are a 3 bit counter clocked by bit 4.

4.12. Call and orientation registers

As described in Section 4.8, two consecutive instruction words with MSB set are interpreted as a CALL instruction, where the 10-bit operand is formed from the least significant 5 bits of each word. When a single instruction word with MSB set is encountered, it is interpreted as an ORIENT instruction and the least significant 5 bits of the instruction word are used to specify the orientation of parts.

When an instruction word with MSB set is encountered, the 5 least significant bits of the word are stored in *call register A*. When the next instruction word is executed and the MSB is not set, then the contents of *call register A* are loaded into the *orientation register*. If the MSB is set, then the 5 least significant bits of this word are stored in *call register B*, and

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

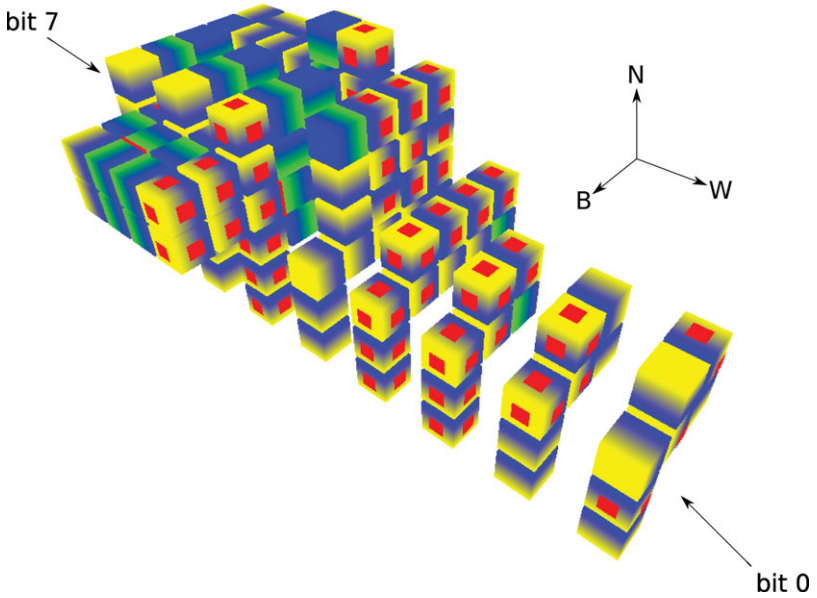


Fig. 26. A graphical representation of the *memory address counter*.

a signal is sent to the *program counter* and *call stack* which causes the *program counter* to be pushed onto the *call stack* before the *program counter* is loaded with an address formed from *call registers A* and *B*.

The *call registers* and the *orientation register* have different requirements and hence different designs. The *call registers* must accept signals 1 time unit long and do not need to hold their value for a long time—only for as long as it takes to execute a CALL instruction.

Figure 27 shows a single *call register*. The shaded parts in the register form moveable structures whose position determines the output of each register bit. These structures are not connected to any other part of the register. The register has a *reset* input that will set all of the outputs *O* to *False* and put the register into a state where future *True* inputs to the parts labelled *A* will cause individual bits to be set. After a *True* signal is received at the part labelled *B*, any inputs to the register through parts *A* will no longer have any effect on the outputs; the outputs will effectly be fixed until the next *reset* signal. Note that the geometry of the inputs to this register matches the geometry of the outputs from the memory module so that all bits entering the register are synchronised. Figure 28 shows a graphical representation of the *call register*.

The *orientation register* takes inputs from the outputs of *call register A* and must store its value until the next ORIENT instruction is executed. The *orientation register* can be implemented in a straightforward way using flip-flops as described in Fig. 4.

4.13. Programming the SRPC

Construction programs for the SRPC are written using the instruction set described in Section 4.8. Programs are assembled and then written into memory modules using a function written for this purpose as part of the suite of classes described in Section 3.8. The construction program required for self-replication fits into 8,191 words. The complete construction program can be found in the supplementary material for this paper.

The child SRPC is constructed in a state where it is about to begin collecting parts, as though it had just executed a GATHER instruction. The parent machine activates the child by injecting a signal into the *detect* mechanism which causes the child to begin moving in a *WEST* direction looking for parts.

The child SRPC is functionally equivalent to the parent machine, but there are some minor differences between parent machine and child in several places. These differences were found to be necessary after the parent machine had been designed when the program for the construction of the child machine was being written. Where necessary, connections between subsystems within the child machine are made using carefully positioned *fuse* parts. In several places, signal paths which are made from *wire* parts in the parent machine are replaced with *nor* parts in the child machine to prevent the storage areas from becoming exhausted during construction. For the same reason, there are several places in the machine where parts that are performing a structural role, in which the part type does not matter, differ between the parent machine and the child machine. No attempt was made to make sure that the precise pattern of part connectivity across the whole machine is identical in the parent and the child. Generally, all neighbouring parts in the parent machine are connected unless the operation of a mechanism requires that they are not. In the child machine, the connectivity pattern is such that any two parts that are indirectly connected in the parent machine will also be indirectly connected in the child machine, but the pattern of direct connectivity may differ.

To help reduce the size of the construction program, the same subroutine is used for constructing most of every *storage mechanism*. The disadvantage of this is that all six *storage mechanisms* in the child machine end up containing 26 *nor* parts which have to be flushed out of the child machine before construction can begin. For this reason, the child machine has some parts protruding *BACK* by 4 units

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

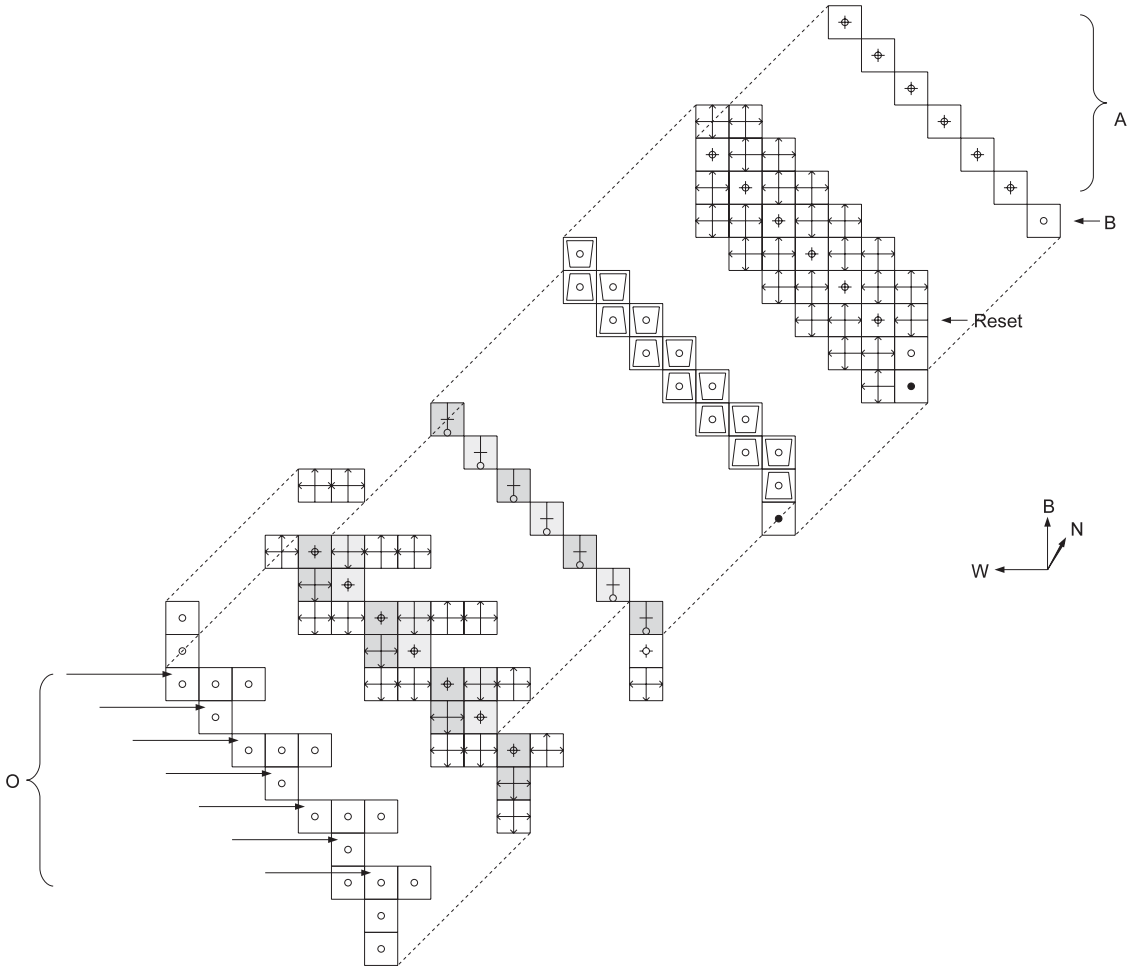


Fig. 27. A *call register*.

from the *NORTH* of the memory—these are the flushed parts.

It would be possible to propagate these small differences between the child machine and parent back into the parent machine but this would be time consuming. These differences do not affect the function of the machine: the machine that the child machine constructs for a given construction program is identical to that constructed by the parent machine. Therefore, the child machine and the grandchild machine are identical.

4.14. Statistics and validation

The number of parts in the parent machine in its starting configuration in which all part storage mechanisms are full

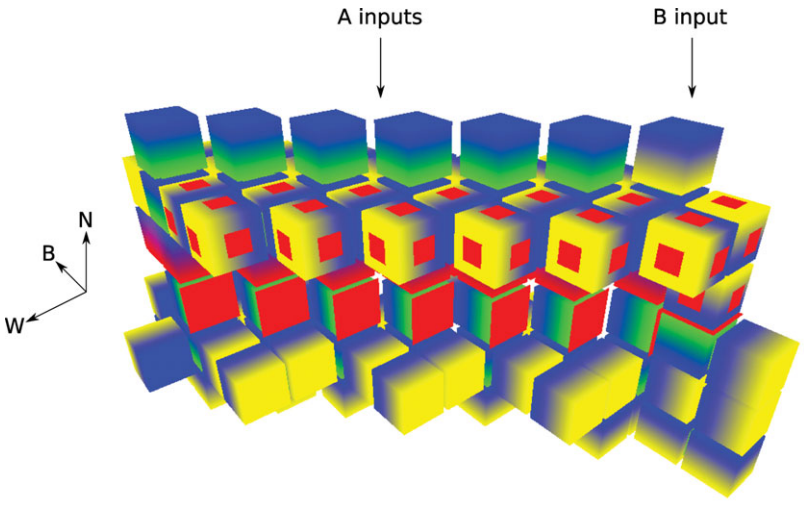


Fig. 28. A graphical representation of a *call register*.

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

Table VII. Number of each type of part used.

Part type	Number used
Wire	54,917
Nor	3,896
Slide	897
Fuse	238
Unfuse	227
Rotate	177

is 60,352. Table VII shows how many parts of each type are used.

The correctness of the design of the machine and of the construction program were validated in three different steps. Firstly, the operation of the *sorter* was validated by programming the SRPC to execute a single GATHER instruction and placing to the *WEST* of the input orifice of the *detect* mechanism a collection of parts containing every type of part in every possible orientation. The machine was able to correctly classify every part that it encountered.

Secondly, a full simulation of an SRPC constructing a replica machine was carried out. In order to minimise the simulation time required for this, the initial environment of the machine contained parts laid out in the *SF* orientation in the order that they would be required. The full simulation takes 224 million time units and runs in about 10 days on a typical desktop computer available in 2009.

The second step was repeated with the child machine in place of the parent machine and then a comparison was made to verify that the machine that the child constructed (i.e., the grandchild machine) was identical to the child machine.

All three validation steps were completed successfully. All of the input files required to validate the design along with the output files containing the results of simulations are available in the supplementary material for this paper. The supplementary material also contains an example of the machine constructing a different machine (a 10 bit asynchrounous counter).

5. Conclusion

From the point of view of physical realism CA environments are underconstrained since they permit the creation of parts from empty space. As was discussed in Section 2.16, adding the constraint that parts cannot be created from nowhere leads to the necessity of being able to move parts around in order to carry out construction operations which in turn leads to a requirement for a facility for connecting parts together so that mobile structures will not fall apart when they move.

Adding this constraint also re-opens some of the questions posed by von Neumann about construction and self-replication. The following questions have been answered positively for certain CA environments, but have not yet been fully answered for physical environments.

1. Can a machine construct any other machine that it is given a description of?
2. Can such a machine construct a duplicate of itself?

Imagine that despite the efforts of von Neumann and subsequent researchers, neither of these questions had yet been answered for CA environments supporting simple component parts, or that a negative answer had been found. This would seriously deter research into physical self-replicating constructing systems. Similarly if it were found to be difficult or impossible to answer these questions positively in a kinematic environment supporting simple parts then this would increase doubt as to whether region G of Fig. 1 could be reached.

The CBlocks3D kinematic environment supports parts chosen to be as simple as possible; it can be regarded as a realisation of the kinematic system originally proposed by von Neumann.[7] This paper answers both of the above questions affirmatively within this environment. In attempting to reach region G travelling from the abstract end of the physical realism axis of Fig. 1 no barrier has yet been encountered.

There are several directions in which this research can be taken forward, listed below.

1. The number of parts in the machine (60,352) can probably be reduced somewhat by revisiting the design of each subsystem of the machine.
2. The physical realism of the environment could be increased by using a more realistic set of laws of motion.
3. The construction program for the machine is written manually. This is a laborious process which could possibly be fully or partially automated.
4. The program length and part storage capacity of the machine have an upper limit and are not extensible (without altering the architecture of the machine). It may be possible to devise an alternative architecture in which the part storage areas can store an unlimited number of parts and in which the program can be any finite length.
5. The machine could make more use of kinematic/mechanical logic. Parts of the address decoder, call stack and call registers depend upon the kinematic features of the environment. It would be interesting to see how much of the machine's controlling logic could be implemented kinematically.

Supplementary material for this paper is available online at: http://www.srm.org.uk/cblocks3d/supplementary.zip

Acknowledgments

This work is an abridged version of chapters 2, 4 and 5 of my PhD thesis 'Self-Replication, Construction and Computation', undertaken with the Department of Physics and Astronomy, Open University, UK between 2004 and 2009.

References

1. M. Bedau (ed.), *Artificial Life* (MIT Press) http://www.mitpressjournals.org/loi/artl (visited on 7th March 2010).
2. D. Kirschner, Y. Iwasa and L. Wolpert (eds.), *J. Theor. Biol.* Elsevier, http://www.elsevier.com/locate/issn/0022-5193 (visited on 7th March 2010).

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press

3. R. A. Freitas and W. P. Gilbreath, "Advanced Automation for Space Missions," (NASA Conference Publications) CP-2255 (N83-15348) (1982), http://www.islandone.org/MMSG/aasm/ (visited on 7th March 2010).
4. M. Sipper, "Fifty years of research on self-replication: An overview," *Artif. Life* **4**(3), 237–257 (1998).
5. R. A. Freitas and R. C. Merkle, *Kinematic Self-Replicating Machines* (Landes Bioscience, Georgetown, Texas, 2004), http://www.molecularassembler.com/KSRM.htm (visited on 7th March 2010).
6. B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts and J. D. Watson, *Molecular Biology of the Cell*, 3rd ed. (Garland Publishing Inc, 1994), pp. 863–946.
7. J. von Neumann and A. W. Burks, *Theory of Self-Reproducing Automata* (University of Illinois Press, Urbana, Illinois, 1966).
8. U. Pesavento, "An implementatin of von Neumann's self-reproducing machine," *Artif. Life* **2**(4), 337–354 (1995).
9. R. Nobili, "The cellular automata of John von Neumann," http://www.pd.infn.it/rnobili/wjvn/index.htm (visited on 7th March 2010).
10. W. R. Buckley, "Signal Crossing Solutions in von Neumann self-replicating cellular automata," *Automata* (Luniver Press, Frome, UK, 2008) pp. 453–501.
11. J. von Neumann, "Re-evaluation of the problems of complicated automata – problems of hierarchy and evolution (Fifth Illinois Lecture – December 1949)," Papers of John von Neumann on Computing and Computer Theory 477–490, MIT Press (1987).
12. B. McMullin, "John von Neumann and the Evolutionary Growth of Complexity: Looking Backwards, Looking Forwards. . .," Artificial Life VII: Proceedings of the Seventh International Conference 467–476, MIT Press (2000), http://www.eeng.dcu.ie/ alife/bmcm-2000-01 (visited on 7th March 2010).
13. C. G. Langton, "Self-reproduction in cellular automata," *Physica* D **10**, 135–144 (1984).
14. J. W. Thatcher, "Universality in the von Neumann cellular model : Technical Report 03105-30-T," College of Engineering – Technical Reports, University of Michigan (1964), http://hdl.handle.net/2027.42/7923 (visited on 7th March 2010).
15. C. Y. Lee, "A Turing Machine that prints its own code script," *Proceedings of the Symposium on the Mathematical Theory of Automata*, Polytechnic Press, Brooklyn, New York (1963) pp. 155–164.
16. R. Laing, Automaton Self-Reference *PhD Thesis* (State University of New York, 1978), http://hdl.handle.net/2027.42/6125 (visited on 7th March 2010).
17. L. S. Penrose and R. Penrose, "A self-reproducing analogue," *Nature* **179**(4571), 1183 (1957).
18. L. S. Penrose, "Self-reproducing machines," *Sci. Am.* **200**(6), 105–114 (1959).
19. S. Griffith, D. Goldwater and J. M. Jacobson, "Robotics: Self-replication from random parts," *Nature* **437**, 636 (2005).
20. A. Smith, P. Turney and R. Ewaschuk, "Self-replicating machines in continuous space with virtual physics," *Artif. Life* **9**, 21–40 (2003).
21. T. Hutton, "Evolvable self-replicating molecules in an artificial chemistry," *Artif. Life* **8**(4), 341–356 (2002).
22. K. Lee, M. Moses and G. S. Chirikjian, "Robotic self-replication in structured environments: Physical demonstrations and complexity measures," *Int. J. Robot. Res.* **27**(3–4), 387–401 (2008), http://ijr.sagepub.com/cgi/content/abstract/27/3-4/387 (visited on March 7, 2010).
23. R. Ewaschuk and P. Turney, "Self-replication and self-assembly for manufacturing," *Artif. Life* **12**, 411–433 (2006).
24. G. Tempesti, "A New Self-Reproducing Cellular Automaton Capable of Construction and Computation," *Proceedingsof the Third European Conference on Artificial Life*, Springer (1995) pp. 555–563.
25. P. A. Carr and M. C. Jewett, "Genome engineering," *Nature Biotechnol.* **27**(12), 1151–1162 (2009).
26. A. W. Burks, *Essays on Cellular Automata* (University of Illinois Press, Urbana, IL, 1970).
27. M. Moses, A Physical Prototype of a Self-Replicating Universal Constructor, *Masters Thesis* (Department of Mechanical Engineering, University of New Mexico, Albuquerque, NM, 2001).
28. V. Zykov, E. Mytilinaios, B. Adams and H. Lipson, "Self-reproducing machines," *Nature* **435**, 163–164 (2005).
29. A. Bowyer, "The Self-replicating Rapid Prototyper – Manufacturing for the masses," *Proceedings of the 8th National Conference on Rapid Design, Prototyping and Manufacturing* Centre for Rapid Design and Manufacture, High Wycombe, June 2007.
30. K. E. Drexler, *Engines of Creation: The Coming Era of Nanotechnology* (Anchor Press/Doubleday, New York, 1986).
31. K. E. Drexler, *Nanosystems: Molecular Machinery, Manufacturing and Computation* (John Wiley and Sons, New York, 1992).
32. W. M. Stevens, "Simulating self replicating machines," *J. Intell. Robot. Syst.* **49**(2), 135–150 (2007).
33. M. A. Arbib, "Machines which Compute and Construct," **In:** *Theories of Abstract Automata* (Prentice-Hall, Englewood Cliffs, New Jersey, 1969) pp. 355–361.
34. W. Gosper, "Exploiting regularities in large cellular spaces," *Physica D.* **10**(1–2), 75–80 (1984).
35. W. M. Stevens, "Parts closure in a kinematic self-replicating programmable constructor," *Artif. Life Robot.* **13**(2), 508–511 (2009).

https://doi.org/10.1017/S0263574710000688 Published online by Cambridge University Press