The Open University

Department of Physics and Astronomy

SELF-REPLICATION, CONSTRUCTION AND COMPUTATION

William Michael Stevens
BSc(Hons.) University of Kent, 1998

Submitted for the Degree of

Doctor of Philosophy

28th October 2009

Abstract

Research into autonomous constructing systems capable of constructing duplicates of themselves has focused either on highly abstract logical models, such as cellular automata, or on physical systems that are deliberately simplified so as to make the problem more tractable. There are also several system-level proposals for physical self-replicating manufacturing systems.

While cellular automata are adequate for modelling the control and information processing aspects of a self-replicating system, these models do not contain any notion of material parts and are poor at modelling those features of a system that depend upon the motion and connectivity of its components.

Physical models of systems with thousands of parts have the disadvantage that they are expensive and time consuming to develop and build.

To overcome these limitations a simulation environment is presented at a level of abstraction that intrinsically models motion and connectivity, and in which parts can be neither created nor destroyed nor transformed into other parts. This level of abstraction lies somewhere between cellular automata and physical systems. Component parts within this environment are designed to be as simple as possible.

A self-replicating programmable constructing machine has been implemented in this environment. The machine takes a disorganised collection of parts as its input and constructs machines from these parts. As a special case it can be programmed to construct a duplicate of itself. The machine is made from 59,615 component parts of which 49,152 make up the machine's memory.

This is the first demonstration of a self-replicating programmable constructing machine in an environment more physically realistic than cellular automata and in which component parts are deliberately chosen so as to be as simple as possible.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Thanks to my family, friends and colleagues for support and encouragement during the past six years. Thanks to my supervisors Professor Nigel Mason and Professor Uwe Grimm for exactly the right amount of supervision.

Chapter 1

Introduction

1.1 Self-replication

The idea of a machine making another machine similar to itself may seem unintuitive or even paradoxical. Our everyday experience of machines that manufacture objects is that the manufactured objects are a great deal simpler than the machine that made them.

Until recent times the only systems known to be capable of producing other systems like themselves were living organisms. Until the nineteenth century many people held the view that living organisms were made from a different type of substance from non-living things and that it was this substance that somehow accounted for some of the seemingly inexplicable properties and abilities of living things, such as their ability to respond to their environment and to grow and reproduce. By the beginning of the twentieth century it was known that living things are composed of the same type of matter as non-living things and it became widely accepted within the scientific community that all of the processes that occur within living things are mechanistic. Coupled with the increasing complexity of man-made machines this understanding led to the recognition that there are general principles of organization and control that apply to both machines and living things. The field of cybernetics arose in response to this [72].

During the second half of the twentieth century it became possible to explain at the molecular level how living things are able to give rise to other living things. An understanding of the process led to research into how the molecular mechanisms found in living things arose, whether the process could be controlled and engineered, and whether a similar process could be implemented in a non-biological machine.

1.2 Open problems related to self-replicating systems

There are a number of open problems related to self-replicating systems. The most important of these are stated here in order to clarify which of them this work attempts to answer and which it does not.

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?

4. Can artificial environments be designed in which self-replicating systems capable of evolution can emerge spontaneously?

These questions are among those that the fields of Artificial Life and Theoretical Biology seek to answer [6], [32].

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 work seeks an answer to 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, some of the early theoretical work on self-replication demonstrated that artificial self-replication is not a logical impossibility, and showed how self-replicating systems are capable in principle of growing in complexity from one generation to another. This early work has implications for questions from both groups.

As a second example of the overlap between the two groups of problems, 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 [22]. 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.

1.3 Terminology

The terms below are used throughout this thesis.

Self-replication: The act of a system causing a duplicate of itself to form.

Self-reproduction: Some researchers have reserved this term for the process of biological self-reproduction or for self-replication with less than perfect fidelity. Others have regarded the term as a less general term than self-replication in which all of the processes involved in the replication of a system take place within the system itself. For example according to this usage a virus could be called a self-replicating system but not a self-reproducing system. However most researchers have regarded this term as being synonymous with self-replication. This is the practice used in this thesis.

Construction: The action of forming by the putting together of parts.

Programmable constructing system: A system that is capable of being programmed to construct objects.

Self-replicating programmable constructing system: A system that is capable of being programmed to construct objects and which is capable of constructing a duplicate of itself.

1.4 Thesis structure

Chapter 2 reviews previous work on self-replicating programmable constructing systems. Chapter 3 describes two pieces of exploratory work that were undertaken shortly before the

formal commencement of the period of study that led to this thesis. The work on kinematic automata in Chapter 3 helped to establish which direction this research should proceed in, and led to the kinematic environment described in Chapter 4. This environment is used for the self-replicating programmable constructor described in Chapter 5.

Chapter 6 contains unanticipated work that emerged as I became more familiar with kinematic environments. This was done in parallel with the work in Chapter 5, but can also be seen as the beginnings of an answer to a question about how the result of Chapter 5 can be simplified.

Chapter 7 summarises the main results of this thesis and lays out some directions for further research.

The CD attached to this thesis contains software for simulating the work of Chapters 3,5 and 6.

Chapter 2

Review of 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 Freitas and Gilbreath's report on a 1980 NASA study of an application of self-replicating systems for space missions [22]. Sipper [55] completed a fifty-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' [24] contains a comprehensive review section (part of which borrows from that of Laing) covering all kinds of self-replicating systems but in particular focusing on physical self-replicating systems.

This review has some overlap with these earlier reviews but a different and more specific focus. The focus is on self-replicating programmable constructing machines as defined in section 1.3. Characteristic examples of other self-replicating systems will be described, but only to point out how they differ from programmable constructing machines, or how they could be extended so as to become more like programmable constructing machines.

The level of attention given to each work depends on its relevance to this thesis. So for example, a detailed description is given of the cellular automaton that von Neumann used for his programmable constructing machine [70]. The related work of Thatcher [65] is also examined. Considerable attention is paid to the works of Moses [44], Zykov et al [74] and Bowyer [9], since these are the most significant attempts to physically realise programmable constructing machines capable of self-replication. Less attention is given

to template-based replication schemes and loop-based replication schemes that have little possibility of physical realisation or little constructive capability, and little attention is given to proposals that lack quantitative definition unless, as in the case of Drexler's early work [17], the proposal brings some unique insights.

Since some of the systems described in section 2.5 were conceived in the context of a particular application, these applications are described alongside the corresponding systems.

2.1 Self-replication in living systems

An outline of the process by which the simplest prokaryotic cells replicate is shown in Figure 2.1

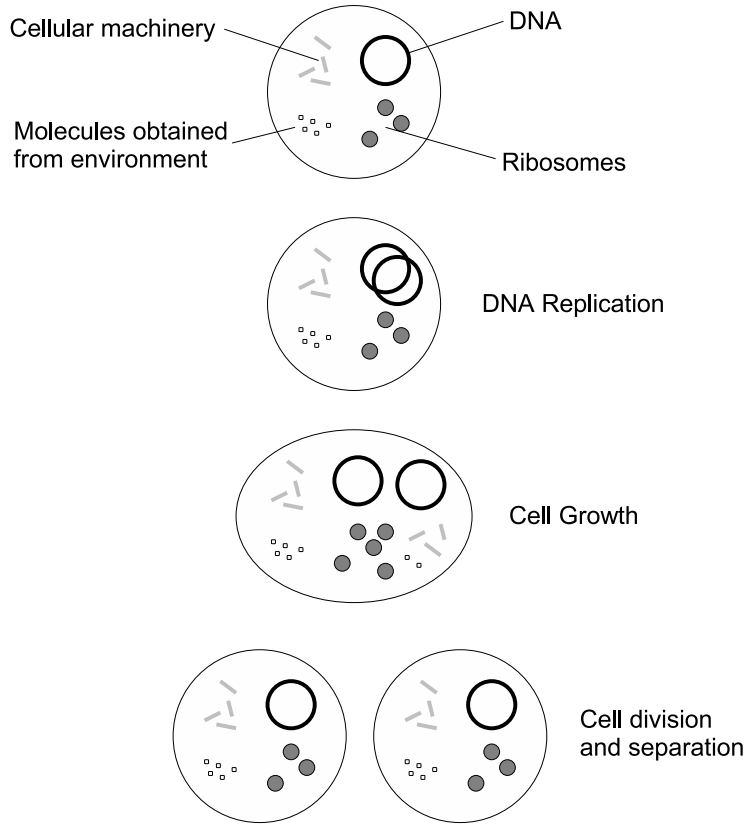


Figure 2.1. The replication process in prokaryotic cells.

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 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. The general features of the replication process in cells were described by Wilson in 1895 [73], several decades before the detailed molecular structure of the material constituents involved was deduced. For example, Wilson was able to write the following hypothesis about inheritance in cells:

> ... chromatin is ... nuclein, which analysis shows ... to be composed of a nucleic acid, a complex organic acid rich in phosphorus. And thus we reach the remarkable conclusion that inheritance may perhaps be affected by the physical transmission of a particular chemical compound from parent to offspring

An up-to-date account of the process can be found in [2].

Some modern bacteria are thought to resemble the earliest forms of life that existed on Earth [40]. They are able to take in simple, abundant molecules consisting of a few atoms each as raw materials and then use these to maintain their own structure and to grow and divide. When one considers individual atoms and the laws that govern their structure and their interaction with other atoms, it is not at all obvious that these simple particles can be joined together to make complex structures that are capable of manipulating and marshalling the very things that they are made of.

How living things arose from non-living things is not known. There are several different plausible hypotheses for the process, for example the RNA world hypothesis [3], and Cairns-Smith's hypothesis that organic life evolved from inorganic self-replicating clay crystals [13].

What we can say with confidence is that the great complexity and diversity that we see in the living world today arose through natural selection acting upon variations in living organisms caused by variations in information encoded in genetic material.

As organisms became more complex, the range of synthetic pathways employed within cells grew, as did the degree of co-operation between individual cells.

An interesting question that will be touched on in section 2.4.3 is the question of whether artificial self-replicating systems can improve upon some of the capabilities of natural self-replicating systems. The particular capabilities that this thesis is concerned

with are the programmability of a system and the range of structures that a system is capable of constructing. Putting the question in the context of this thesis, we can ask how much the evolutionary origins of living cells constrain the ease with which they can be programmed, and the range of substances they can be programmed to make.

2.2 Von Neumann's work

2.2.1 An abstract self-replication process

In the late 1940s, before the discovery of the structure of the genetic material and before the processes by which information encoded in the genetic material gives rise to cellular structure and function were understood, von Neumann [70] described an abstract self-replication process that is remarkably similar to the process that was eventually found to be taking place within cells. In outline, von Neumann's process is as follows:

- There is a general constructing automaton A which can construct any automaton X given a description $\phi(X)$ of X. Von Neumann proposed that for the sake of simplicity, $\phi(X)$ should be a linear description, perhaps in binary form.
- There is a description-copying automaton B which can construct a copy of any description $\phi(X)$.
- There is a control automaton C which first of all activates A, then activates B.
- A, B, C and $\phi(X)$ can be composed into a single automaton $(A + B + C) + \phi(X)$.
- The action of automaton A in $(A+B+C)+\phi(X)$ leads to $(A+B+C)+\phi(X)+X$
- The action of automaton B in $(A + B + C) + \phi(X) + X$ leads to $(A + B + C) + \phi(X) + X + \phi(X)$
- By setting X to $A + B + C$, we achieve self-replication.

This is a slight simplification of the process described in [70], in which automaton A consumes the description $\phi(X)$ that it operates upon and so B is invoked twice in order to make three copies of $\phi(X)$.

The parallels with self-replication in living cells are as follows:

- The description $\phi(X)$ recorded in a linear form is analogous to the genome, stored in linear DNA.

- The description-copying automaton B is analogous to the collection of about a dozen proteins that are involved in DNA replication.

- The general constructing automaton A is analogous to the ribosome and associated machinery, which between them attempt to construct proteins according to any description given to them.

- The controlling automaton C is analogous to the machinery that regulates the cell cycle in a living cell.

Although von Neumann's process was worked out before the precise details of cellular reproduction were known, his writings indicate that his thoughts on the subject were informed by what was known at the time about cellular reproduction, so it is perhaps not very surprising that similarities exist between the two processes.

2.2.2 Von Neumann's 5 questions

Von Neumann's work on self-replication must be understood in the context in which it was undertaken. In reference [70], it is clear that von Neumann was interested in answering the 5 questions quoted below:

> (A) Logical universality. When is a class of automata logically universal, i.e., able to perform all those logical operations that are at all performable with finite (but arbitrarily extensive) means? Also, with what additional — variable but in the essential respects standard — attachments is a single automaton logically universal?
>
> (B) Constructability. Can an automaton be constructed, i.e., assembled and built from appropriately defined "raw materials," by another automaton? Or, starting from the other end and extending the question, what class of automata can be constructed by one, suitably given, automaton? The variable, but essentially standard attachments to the latter, in the sense of the second question of (A) may here be permitted.
>
> (C) Construction-universality. Making the second question of (B) more specific, can any one, suitably given, automaton be construction-universal, i.e., be able to construct in the sense of question (B) (with suitable, but essentially standard, attachments) every other automaton?

(D) Self-reproduction. Narrowing question (C), can any automaton construct other automata that are exactly like it? Can it be made, in addition, to perform further tasks, e.g., also construct certain other, prescribed automata?

(E) Evolution. Combining questions (C) and (D), can the construction of automata by automata progress from simpler types to increasingly complicated types? Also, assuming some suitable definition of "efficiency," can this evolution go from less efficient to more efficient automata?

At the time that von Neumann undertook his work, the answers to both questions of (A) were known. Turing machines are an abstract class of automaton that answer the first part of this question, and the universal Turing machine answers the second part [66].

Von Neumann went on to answer questions B to E. Although his work was not finished at the time of his death in 1957 the manuscripts that he left contained enough information for Burks to complete that part of the work that was concerned with the design of a self-replicating programmable constructing automaton.

2.2.3 A cellular automaton model of a self-replicating programmable constructor

Questions B to E were answered by devising a cellular automaton (CA) with transition rules designed to support component parts which allow the implementation of signalling pathways and logical operations, as well as construction pathways and construction operations. Figure 2.2 show how 29 CA states are used for five types of component part, including an Unexcitable component which 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 which turn a component back into the Unexcitable component.

Transmission components T behave like directional OR gates with three inputs and one output. A Transmission component may be quiescent or excited (excited states are those having dots in Figure 2.2). When a Transmission component p has a neighbouring excited Transmission component q, and the output of q lies against an input of p, p will be excited one time unit later. p will also be excited one time unit later if an excited Confluent component lies against an input of p. If neither of these conditions are met, p will be quiescent one time unit later. 8 CA states are used for the Transmission component.

Confluent components C are directionless and may also be quiescent or excited. A

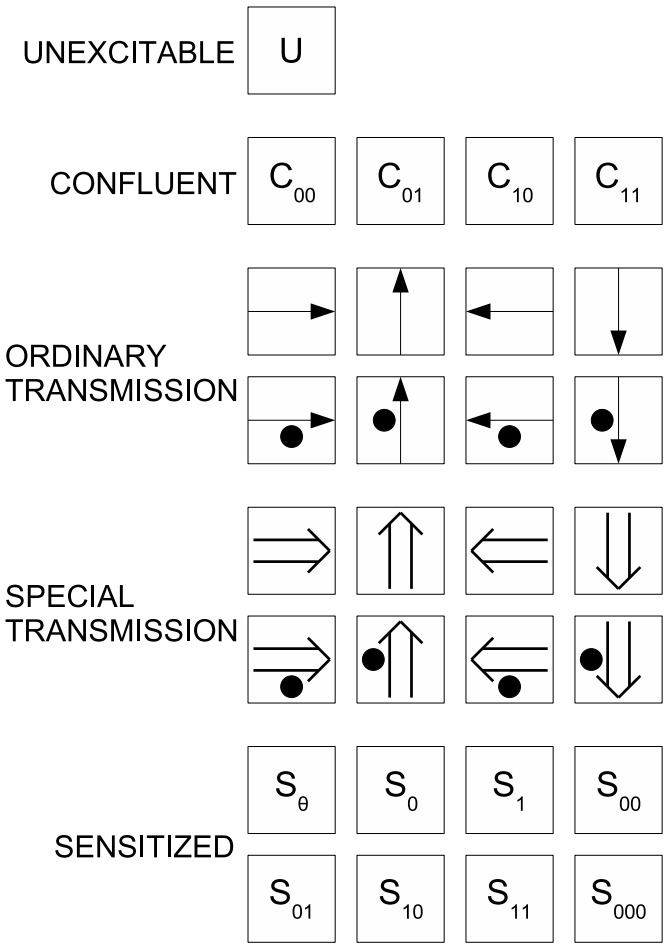


Figure 2.2. The 29 states of von Neumann's cellular automaton.

Confluent component can be thought of as a delaying AND gate with between one and three inputs. A Confluent component p will become excited two time units later if at least one of its neighbours is an excited Transmission component with output pointing at p, so long as none of its neighbours are quiescent Transmission components with output pointing at p. Otherwise p will be quiescent two time units later. Because the neighbours of a Confluent component determine whether or not it will be excited two time units later it can be used to introduce a delay of two time units into the propagation of an excited state along a pathway (whereas a Transmission component introduces a delay of one time unit). 4 CA states are used for the Confluent component. In Figure 2.2 and elsewhere, C_{xy} denotes a Confluent component where $x = 1$ means that the component is currently excited and $x = 0$ means that it is not currently excited. Similarly $y = 1$ means that the component will be excited in the next time unit and $y = 0$ means that it will not be

excited in the next time unit.

Special Transmission components T' have similar behaviour to Transmission components in that they can be used to propagate stimuli. The main purpose of a Special Transmission component is to apply a stimulus to a Transmission or Confluent component that is no longer needed in a particular location. This causes the component to turn into an Unexcitable component. Special Transmission components may be excited by other Special Transmission components or by an excited Confluent component. If a Special Transmission component needs to be turned into Unexcitable component this can be done by stimulating it with a Transmission component.

The Unexcitable component U does nothing unless stimulated by a Transmission or Special Transmission component, in which case it turns into the Sensitised component S_θ.

Sensitised components are the means by which an Unexcitable component can be turned into other components in order to effect construction operations. The Sensitised component S_θ may be stimulated by a sequence of signals in order to transform it into the quiescent form of the Transmission, Confluent or Special Transmission components. The CA states used for the Sensitised component are denoted S_σ, where $\sigma \in \{\theta, 0, 1, 00, 01, 10, 11, 000\}$. After one time unit, Sensitised state S_σ becomes $S_{\sigma 1}$ or $S_{\sigma 0}$ depending on whether or not it is stimulated. The states S_{0000}, S_{0001}, S_{001}, S_{010}, S_{011}, S_{100}, S_{101}, S_{110}, S_{111} are the 9 quiescent states of the Transmission, Confluent and Special Transmission components described above.

Von Neumann chose these components with the expectation that they would possess logical universality in the sense of question A, which was indeed shown to be the case, and also that they might permit the implementation of constructing automata, which when fed appropriate signals might cause the creation of patterns in a previously empty (Unexcitable) region of the CA space. Von Neumann considered only the construction of quiescent patterns which would be entirely inactive until excited by an initiating stimulus.

Von Neumann devised several simple logical 'organs' for his computing and constructing automata which were then combined to make more complex subsystems. Figures 2.3, 2.4 and 2.5 show three such organs. (The organ of Figure 2.4 is not von Neumann's design, but a later design of J.E. Gorman).

Figure 2.6 shows a schematic diagram of von Neumann's self-replicating programmable constructor, with the main subsystems labelled. With reference to the process outlined in section 2.2.1 the Instruction Tape corresponds to $\phi(X)$, the Control Logic corresponds to automaton C. Between them the Construction Logic, Arm Control and Construction Arm correspond to automaton B and the Tape Copying Logic, Arm Control and Construction

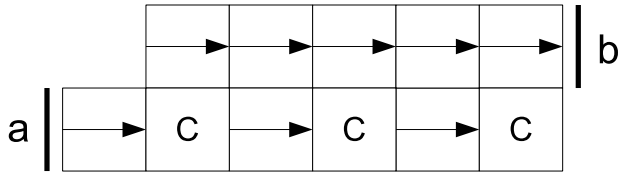


Figure 2.3. A Pulser organ which outputs 111 at b when excited with 1 at a.

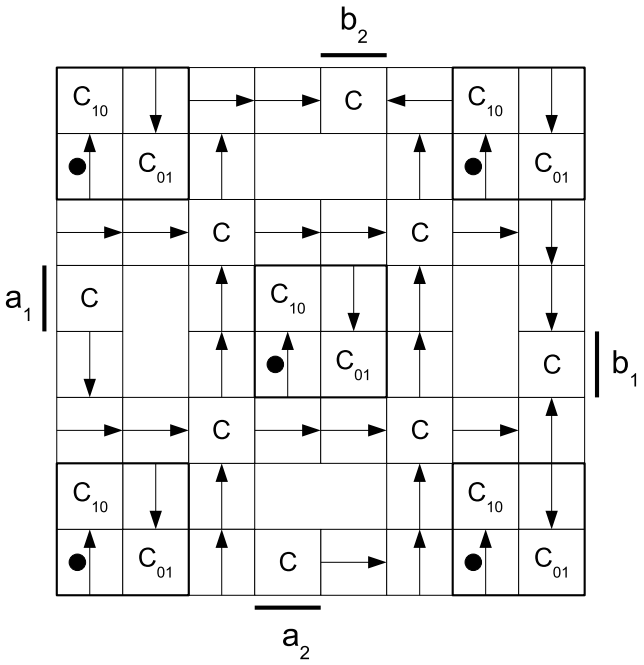


Figure 2.4. An organ which permits sequences of excitations input at a_1 and a_2 to cross over to outputs b_1 and b_2 respectively.

Arm correspond to automaton A. The Construction Arm reaches out from the main body of the machine into an empty region of space and creates a new pattern according to the contents of the Instruction Tape. The Tape Copying logic contains logic for making a copy of the Instruction Tape in the newly created machine.

Von Neumann did not fully specify a cell-by-cell description of a design for a self-replicating programmable constructor. Only recently has this been completed.

In 1995, Nobili and Pesavento succeeded in implementing a version of von Neumann's automaton by augmenting von Neumann's 29 states with 3 additional states for handling signal crossing [49]. This implementation is shown in Figure 2.7. The total number of cells in this implementation is a little over 150,000 (including the instruction tape, not

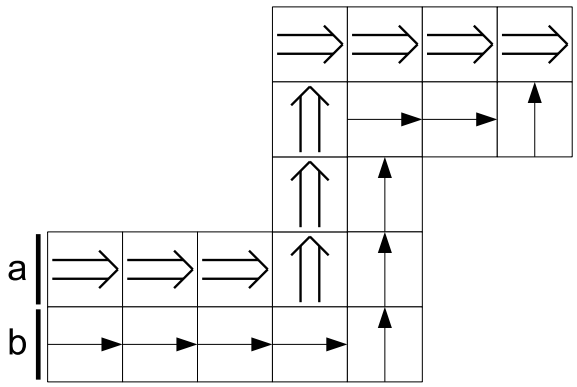


Figure 2.5. The Construction Arm. Appropriate sequences of signals at a and b can cause the arm to grow, turn corners, retract, and transform Unexcitable components into other components.

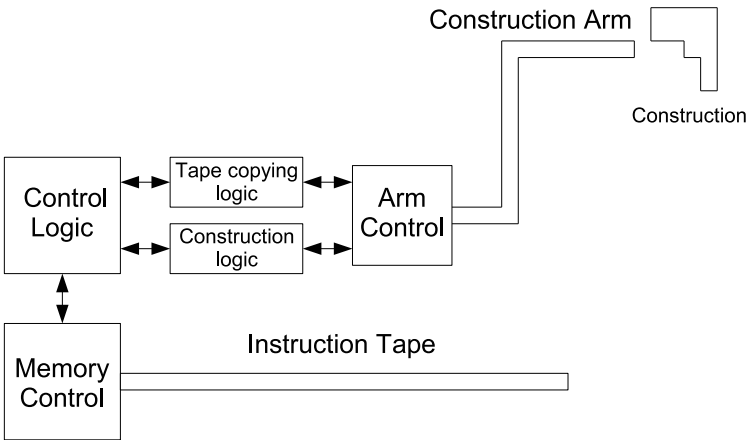


Figure 2.6. A schematic diagram of von Neumann's self-replicating automaton.

shown in Figure 2.7). Nobili improved this to 50,000 cells in 2007 [46] by using run-length encoding for the instruction tape of the machine.

In 2008 Buckley succeeded in implementing and simulating a self-replicating system in von Neumann's environment using the original 29 state rule set [10]. Buckley's implementation uses approximately 310,000 cells.

Von Neumann's self-replicating automaton has sometimes been described as a universal computer-constructor, capable of constructing any specified automaton within the CA environment and also able to perform any specified computation within that environment. However, in the original form that he devised it, it was not. Although von Neumann devised a universal Turing machine in his 29-state CA and although it shares some organs with his universal constructor, his design for a universal constructor did not incorporate

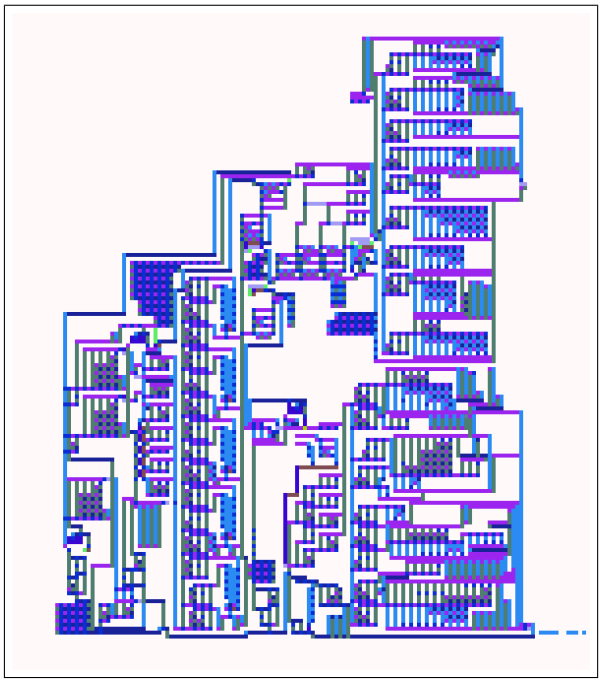


Figure 2.7. A representation of Nobili and Pesavento's implementation of von Neumann's self replicating automaton.

a universal computer. The later works of Thatcher [65] and Codd [14], which have some similarities to von Neumann's design, did incorporate a universal computer as an integral part of the universal constructing automaton for a very good reason that will be explained in section 2.3.1.

In addition to the parallels mentioned in section 2.2.1 between von Neumann's abstract self-replication process and the self-replication process found in living cells, this particular implementation also has in common with living cells that the self-replicating entity is self-contained and topologically continuous.

There are also some significant differences:

- The operation of the machine is entirely deterministic and predictable.
- Information stored on the data tape directly encodes the structure of the machine: there is a one-to-one mapping between a part of the constructed machine and a single region on the tape.
- The machine is not designed to tolerate errors. All parts of the machine are critical to its operation and it has no redundancy nor any ability to detect and correct errors.

In section 2.1.4 of reference [24] Freitas and Merkle 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 (which will be discussed more fully in section 2.4). 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.

A letter that von Neumann wrote to Norbert Wiener in 1946 [69] indicates that he had recently become interested in the problem of self-reproduction in relation to living organisms. The letter hints that even at this early stage of his thinking about the subject he was interested in automata that have a more general constructional capability than is required simply to construct a duplicate automaton.

In reference [68] von Neumann makes it clear that he is aware that there are some simple systems that exhibit a kind of self-reproductive behaviour, and he proposes a definition of self-reproduction that seems to exclude these systems.

> One of the difficulties in defining what one means by self-reproduction is that certain organizations, such as growing crystals, are self-reproductive by any naive definition of self-reproduction, yet nobody is willing to award them the distinction of being self-reproductive. A way around this difficulty is to say that self-reproduction includes the ability to undergo inheritable mutations as well as the ability to make another organism like the original.

McMullin [41] argues that the problem that von Neumann solved by devising the machine described in section 2.2.3 was primarily the problem of how machines can construct other machines, and in particular how machines can give rise to machines more complex than themselves, which was an unsolved problem in nature at the time. McMullin points out that von Neumann's cellular automaton pattern that can be programmed to construct any other pattern, including a duplicate pattern, is a solution to that problem. The simpler self-replicating systems of Langton [37] and Byl [12] do not solve the same problem because these systems cannot construct other systems.

McMullin's interpretation of von Neumann's work is consistent with von Neumann's stated intention of answering questions A to E given on page 9.

Burks appears to misinterpret von Neumann's work since he omits discussing how it relates to question E altogether from his summary in reference [70]. He also alters the 'further tasks' that an automata might carry out in question D from '...also construct certain other, prescribed automata.' to '...can perform the computations of a universal Turing machine...'. Not only does Burks' question D remove the link between questions C and D, but Von Neumann's design does not answer Burks' question D. Burks briefly discusses how von Neumann's design can be modified so as to answer his version of question D, but the question has since been answered by automata much simpler than von Neumann's [27].

This discrepancy between the von Neumann's question D and Burks' question D highlights the fact that the capacity for self-replication is one design requirement or design constraint among many others that a system may have. If the design requirements change, then the range of possible solutions is also likely to change.

2.3 Works directly related to von Neumann's work

Following the completion and publication of von Neumann's work in the 1960s, further research into self-replicating systems embedded in cellular automata led to several other schemas for self-replicating systems.

2.3.1 Thatcher and Codd

Thatcher developed an alternative strategy for self-reproduction in von Neumann's CA environment [65]. Thatcher's development was based upon a self-describing property of Turing machines first proved by Lee [38]. Lee showed that given a universal Turing machine U which accepts descriptions $D(T_n)$ of Turing machines T_n it is possible to find a

Turing machine T_S with the property that when U operates on $D(T_S)$ it outputs a duplicate description $D(T_S)$. We have seen that von Neumann's self-reproducing automaton contained a subsystem dedicated to copying the instruction tape from a parent automaton to its child, and that the controlling subsystem in von Neumann's automaton was not a universal computer. Thatcher realised that if a universal computer is used for the control unit of the automaton then the self-describing property proved by Lee makes 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 self-replicating programmable constructing machines. If it can be shown that a 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.

However, this elegance comes at a cost: the number of elementary parts needed to implement a universal computing automaton is likely to be larger than the number needed to implement a simple translating automaton and a tape-copying automaton. If the length of time that the machine takes to operate is also a consideration then it must be borne in mind that those models of computation that require the least hardware to implement (such as the Turing machine) also turn out to be the slowest. So in order to build a time-efficient universal computing machine the number of elementary parts required is likely to be greater still.

Using Thatcher's approach Codd was able to show in [14] that von Neumann's result could be recapitulated in a simpler 8-state cellular automaton, at the expense of a much larger number of cells — estimated at 100,000,000 [34]. Devore greatly simplified Codd's design, resulting in an implementation using only 94,794 cells [15].

2.3.2 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 [36].

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.

2.4 Simple self-replicating 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 easy in a cellular automaton environment and not very difficult in a physical environment.

In a cellular automaton environment consider a two state system with states p and q, and a rule by which state p transitions to state q whenever any of its neighbours are in state q. In a system containing no q states, no q states will ever appear. However, if the system is seeded with an initial q state then this state 'self-replicates' and spreads throughout the entire universe as quickly as possible.

As a physical analogue of this system imagine a large number of cotton wool balls soaked in a flammable liquid. A flame-proof sack of such balls will remain passive unless seeded by a flaming ball, in which case the flaming ball will 'self-replicate' by propagating its state throughout the sack until all balls are aflame. This example is slightly more complex than the cellular automaton example because an external raw material (oxygen) is also required in order for the state transition to take place, and because the replicating entities have a finite lifetime.

In both of the above cases no information is transferred from parent to child. In the case of the cellular automaton environment the situation becomes no more difficult to arrange however large the number of bits to be transferred: one simply defines a state space large enough to accommodate the number of bits required. In the physical world however this additional requirement does introduce some more complexity.

This requirement was first addressed by Lionel and Roger Penrose, who were among the first to construct physical self-replicating devices. The simplest of their devices [48] consisted of two plywood parts A and B (Figure 2.8).

A collection of these parts can be placed along a line in a random arrangement (Figure 2.9). If the collection is then agitated in one dimension, allowing the parts to jostle against one another, the parts will not interlock with one another unless seeded by an initial configuration (Figure 2.10).

This configuration will cause other pairs of parts to adopt the same configuration. Note that there are two possible configurations for the interlocked pair A and B: that shown in Figure 2.10 and its mirror image. Thus a configuration of parts transfers a single bit of

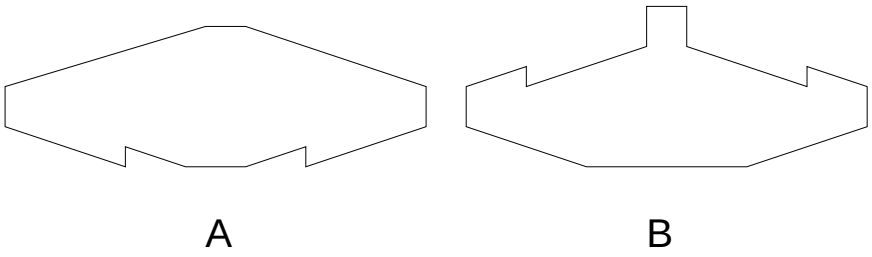


Figure 2.8. The two parts of Penrose's simple self-replicating system.

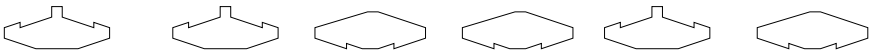


Figure 2.9. A random arrangement of Penrose's parts.

information to its progeny unlike the flaming ball system described above.

2.4.1 Template based systems

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 [47]. Penrose proposed 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 units 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. Figure 2.11 depicts Penrose's concept.

A more recent physical implementation was undertaken by Griffith who devised a system for exploring physical templating processes. Griffith was concerned with building *reliable* working templating systems, rather than simulating such systems. Griffith [26] built a set of programmable units for investigating self-assembly and used them to implement a template-based replication scheme similar to that described by Penrose.

Smith et al [58] and Hutton [28] have developed simulation models of template-based replication in two dimensional spaces.

Figure 2.10. Parts replicating a configuration.

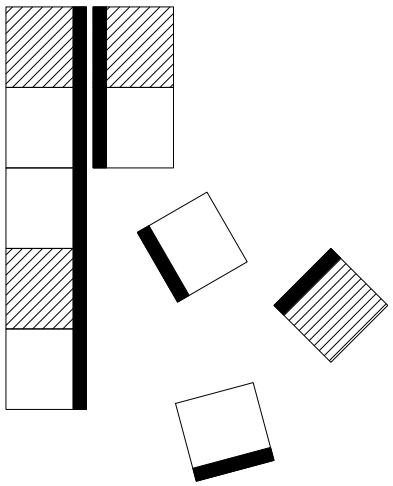


Figure 2.11. Penrose's conception of a template-based self-replicating system.

A systematic analysis of the logic involved in template-based replication and of the energy considerations involved in physical template-based replicating systems has not yet been carried out.

Using LEGO® modules, Chirikjian's research group built a number of simple self-replicating robots [39]. 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.

2.4.2 Langton's self-replicating loops

Taking as his starting point the 8-state cellular automaton devised by Codd and driven towards simplicity, in part by the meagre power of home computers at the time, Langton [37] devised a simple self replicating loop structure embedded in Codd's cellular automaton. 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. Figure 2.12 shows Langton's system after a single replication cycle.

Langton's loop cannot be programmed to build anything other than a replica of itself so it does not have any constructional capability in the sense that the automata of von Neumann and Codd do.

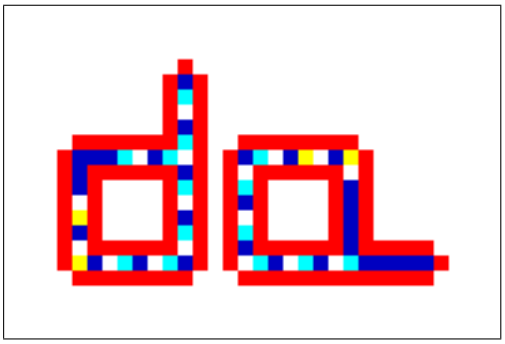


Figure 2.12. Langton's self-replicating loop after a single replication cycle.

Byl [12] was able to design a smaller self-replicating loop than Langton, and Reggia [52] a still smaller one at only five cells but one which still retains the essential features of Langton's loop: circulating instructions controlling a growing tip.

2.4.3 Summary

None of these simple models of self-replication contain anything like a programmable constructor and cannot construct anything except copies of themselves.

Nevertheless it is still worth considering whether such systems can inform the study of self-replicating programmable constructing systems. 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 with a much wider constructional capability.

So perhaps one way to make artificial self-replicating programmable constructing systems is to mimic the path taken by evolution: begin by investigating simple physical 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 Smith et al. where a simple template-based replicating system [58] is later augmented by a templating-folding system so that arbitrary shapes can be constructed [67].

There are however a number of reasons for following a different approach to that found in nature.

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.

For example consider the ribosome. A ribosome is a simple kind of programmable constructor which makes proteins from amino acids. Its construction activity is limited to peptide bond formation between amino acids. Ribosomes construct long chains of amino acids where the sequence of amino acids is specified by a description encoded on RNA. As they are constructed, the amino acid chains fold into a configuration determined by the amino acid sequence. This happens because electrons in molecular subgroups in different parts of the chain interact to cause the chain to fold in a particular way. Often further steps, such as cross linking between different parts of the chain, take place after initial folding but before the protein ends up in its final configuration. Once it has formed the protein then carries out its specific role within the cell.

The amino acid sequence is effectively being used for two different purposes. It is being used for geometrical purposes to give the protein its shape, and it is also being used to give the protein its function by having particular molecular groups in the active sites of the protein. These two purposes are not entirely unrelated, because the action of a protein often depends on its ability to twist and distort as it binds substrate molecules and then returns back to its original conformation once it has completed its operation on substrate molecules. However for at least some proteins much of the structure of the protein exists solely to make sure that the right molecular groups end up in the right places in relation to other molecular groups.

Why have more efficient protein-like molecules not evolved? And why has nothing more elaborate than the ribosome evolved for the construction of proteins? Why do we not see mechanisms in living cells that actively direct and control the folding of macromolecules, thus relieving the macromolecule from the dual role of folding and function? Rather than relying on waiting for random motions and trial and error for the correct amino-acid bearing transfer-RNA to arrive at the correct site in a ribosome, why is there no mechanism for fetching the correct transfer-RNA on demand, perhaps from a buffered store? Would these systems not be more capable or more efficient than the ribosomes that we do see?

One reason may be that such systems could only be expected to occur in living things if there is an evolutionary path towards them. It may be possible to add these features to ribosomes artificially by deliberately redesigning ribosomes to incorporate new features, but unless there is a gradual step-by-step process by which these things can be achieved, we cannot expect that they would occur naturally.

Drexler gives a similar argument when discussing the differences in evolutionary capacity between evolved and designed machines [18].

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

2.5 Physical self-replicating constructors

2.5.1 Systems derived from living systems

Advances in the field of synthetic biology have led to the ability to create some of the components of living cells from scratch. In [29] Jewett and Church describe how they succeeded in making a functioning ribosome in vitro. This research is likely to make it possible to make a synthetic cell containing a metabolism designed for a specific purpose and a genetic code containing only information that is essential for the cell to function.

In [17] Drexler proposed that one of the possible avenues towards being able to manipulate matter precisely at a molecular level is to begin with manipulation and assembly tools derived from biological systems and engineer them so as to increase their functionality and increase the range of materials that they are able to work with. One example of this approach is DNA origami [54], in which a strand of DNA containing 7,176 bases can be programmed to fold-up into any one of a large number of possible configurations.

The two approaches are different. The former approach results in an architecture directly derived from the architecture of a cell. The latter approach seeks to make entirely new kinds of machines from molecular biological components.

At the end of section 2.4.3 it was pointed out that one of the primary constraints that affects the architecture 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.

2.5.2 Von Neumann's kinematic model

Before developing his 29-state cellular space for exploring self-replication and construction von Neumann envisaged a physical system in which he thought it might be possible to construct a self-replicating machine, quoted below from [71]:

> The constructing automaton floats on a surface, surrounded by an unlimited supply of parts. The constructing automaton contains in its memory a 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. In [71], Burks attempted to reconstruct the most detailed description of these parts that von Neumann's gave, quoted below:

> Von Neumann described eight kinds of parts. All seem to have been symbolized with straight lines; inputs and outputs were indicated at the ends and/or the middle. The temporal reference frame was discrete, each element taking a unit of time to respond. It is not clear whether he intended this list to be complete; I suspect that he had not yet made up his mind on this point.
>
> Four of the parts perform logical and information processing operations. A *stimulus organ* receives and transmits stimuli; it receives them disjunctively, that is, it realizes the truth-function "p or q." A *coincidence organ* realizes the truth function "p and not-q." A *stimuli producer* serves as a source of stimuli.
>
> The fifth part is a *rigid member*, from which a rigid frame for an automaton can be constructed. A rigid member does not carry any stimuli; that is, it is an insulated girder. A rigid member may be connected to other rigid members as well as to parts which are not rigid members. These connections are made by a *fusing organ* which, when stimulated, welds or solders two parts together. Presumably the fusing organ is used in the following way. Suppose a point a of one girder is to be joined to point b of another girder. The active or output end of the fusing organ is placed in contact with points a and b. A stimulus into the input end of the fusing organ at time t causes points a and b to be welded together at time $t + 1$. The fusing organ can be withdrawn

> later. Connections may be broken by a *cutting organ* which, when stimulated, unsolders a connection.
>
> The eighth part is a *muscle*, used to produce motion. A muscle is normally rigid. It may be connected to other parts. If stimulated at time t it will contract to length zero by time $t + 1$, keeping all its connections. It will remain contracted as long as it is stimulated.

Kemeny's 1955 article [31] is often cited as containing a description of von Neumann's kinematic model but in fact this article does not describe this model. The article contains two sentences that mention a hypothetical machine capable of making such things as rolls of tape, pencils, erasers, vacuum tubes, motors and so on from raw materials and then constructing a duplicate of itself from these. The article then goes on to describe von Neumann's cellular automaton model of a self-replicating programmable constructor without mentioning the kinematic model at all.

The architecture that von Neumann envisaged for his kinematic self-replicating machine was somewhat similar to that which he later used for his cellular automaton based self-replicating automaton: i.e. a control unit governing the actions of a constructing unit, capable of producing any automaton according to a description provided to it on a linear tape-like memory structure.

Von Neumann expected that the total number of parts required to construct a self-replicating automaton in the kinematic model would run into the millions.

Von Neumann postponed further work on this model after deciding that the cellular automaton model would be easier to reason about and would be sufficient for answering questions A to E of section 2.2. He intended to deal with the logical aspects of self-replication and construction first and then progress to the kinematical, mechanical, chemical and physical aspects.

The quotations below (from [11]) shows that Burks was of the opinion that pursuing a kinematic model is unnecessary:

> It is a difficult problem to develop a complete, precise set of rules for the kinematic system which is at the same time simple and enlightening. Moreover, it is doubtful that anything of much value is contributed by the kinematic or motional capabilities of the system. On the one hand, these kinematic features of the model are too remote from chemistry, physics and mechanics to be of much interest in their own right; while on the other hand, they are

> too remote from problems of organization, control, and logic to contribute to our understanding of these problems.
>
> As far as studies of logic and automata theory are concerned, the cellular system is superior to the kinematic system just because it does not include motion as a basic operation. Motion is not a proper object of study for logic, and as we noted in discussing von Neumann's passage from the kinematic to the cellular system, the motional aspects of the kinematic system complicate it from the point of view of logical analysis.

Burks dismisses systems which model motion as being unnecessary for the *logical* study of automaton and yet there are at least two subsystems of von Neumann's automaton that could 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 the 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 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. The mechanisms which implement these movements, by erasing and recreation, are the only subsystems in the constructor for which the erasing process is required and for which the 8 'Special Transmission' states described in section 2.2 are needed. So we see that despite trying to remove motion from the model, it returns again by the back door in an ill-fitting way.

In stating that any kinematic model would be too remote from chemistry, physics and mechanics to be of much interest in its own right, Burks 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, some of which were described in section 2.4.1.

The systems of Penrose, Griffith, 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 continuous space and discrete 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 arbitrary choices about which concepts to model and how precisely to model them but equally arbitrary 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 (and others) have shown that programmable constructor based self-replication is possible 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 self-replicating programmable constructor but it does not tell us whether there are any other possible 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.

This question cannot be answered using a cellular automaton framework that only models logic and topology, but can be answered using some kind of kinematic model.

2.5.3 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 [44]. 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.

Moses' ultimate ambition with respect to self-replicating systems is to create a machine that can make its own component parts from raw materials and then assemble those components into a replica. His Masters thesis describes his work towards the assembly part of this goal and then goes on to speculate about possible methods for enabling a machine to make its own component parts from raw materials.

Figure 2.13 is a graphical representation of Moses' system with the main subsystems labelled and Figure 2.14 is a photograph of the actual system showing the machine and the replica that it has constructed.

Moses' system is based around a basic polyurethane part shown in Figure 2.15. This

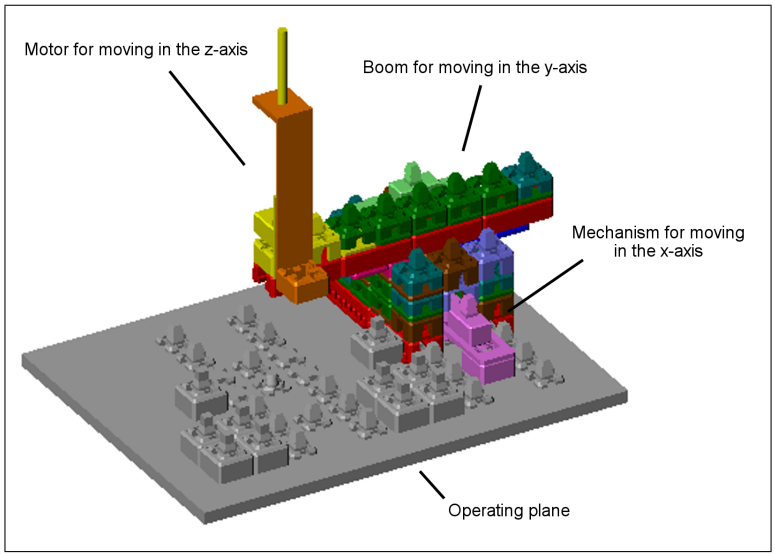


Figure 2.13. Graphical representation of Moses' constructor.

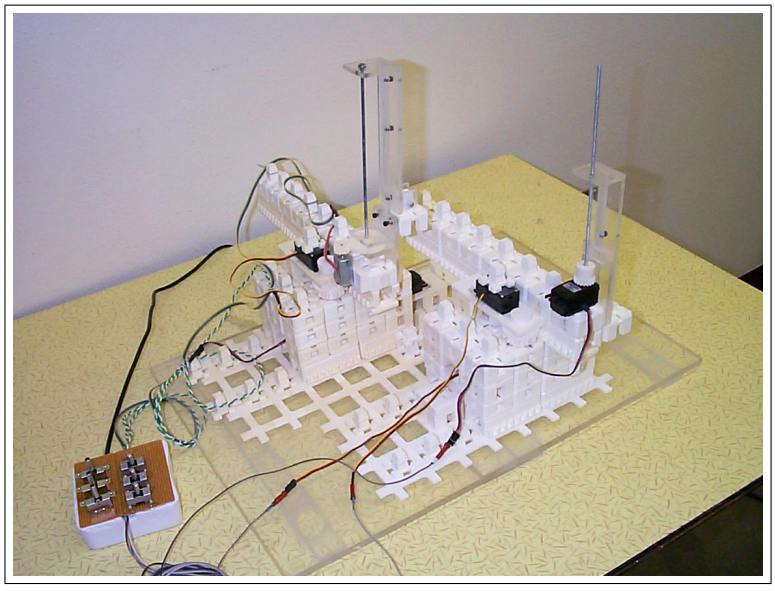


Figure 2.14. A photo of Moses' constructor.

figure shows an exploded view from two different angles. The two separate objects shown in each view are glued together to make a basic part.

The basic part has a snap-tang arrangement for joining parts together and a protrusion which serves both as a handle for picking the part up and as a target for the snap-tangs.

The basic part itself is not used in the constructor. The list below describes other parts derived from the basic part and shows how frequently each part is used in the constructor.

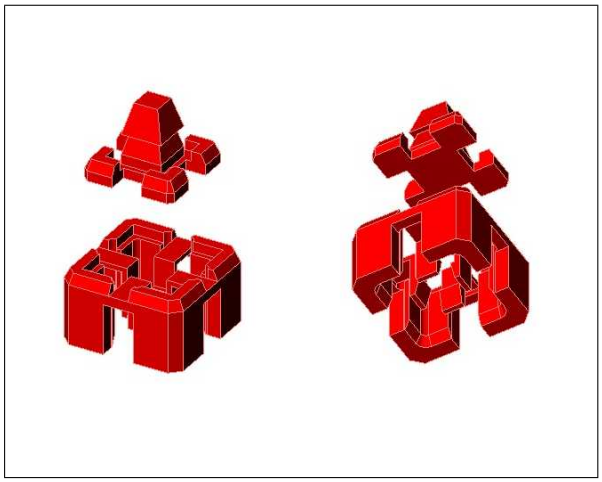


Figure 2.15. A basic part in Moses' system.

Rack: A part roughly the size of two basic parts that has a row of teeth that mesh with the small gear and which can slide back and forth along one axis. Used 8 times.

Motor 1: A part containing a DC motor and a small gear that can be used to drive the rack or the large gear. Used 1 time.

Anchor: A part made from a basic part connected to a tab. It is used to strengthen beams and platforms made from other parts. Used 7 times.

Rail: Performs a similar function to an anchor. Used 13 times.

Cap: Connected to an anchor or used to terminate a chain of rails. Used 8 times.

Rotary cap: Identical to a cap except that the handle is replaced with a rotary handle. Serves as a seat for gears. Used 1 time.

Cross-member: Spans two walls that are spaced one unit apart. Used 3 times.

Cap with support: Similar to a rail, but with the purpose of supporting a sliding beam that travels over the cap. Used 1 time.

Cross-member with support: Similar to a cross-member but with the purpose of supporting a sliding beam that travels over the cross-member. Used 1 time.

Motor 2: The same as motor 1 but with an additional extra basic part. The tangs on the extra basic part help to hold the motor on the operating plane. Used 1 time.

Motor 3: The largest and most complicated part in the set. Provides linear motion in the z direction. Used for retrieving and assembling parts. Used 1 time.

The constructing machine is supplied with parts by the operator. Each part is placed in a storage site. The machine moves its arm 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:

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. In practice the subsystem for generating control signals for a self-replicating constructor is likely to be the largest (if not the most complex) part.

3. Moses' system contains no feedback so, in the absence of a human operator, the system would not know if it had made a mistake during construction.

4. The set of parts contains considerable redundancy and the choice of parts is somewhat arbitrary.

2.5.4 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 [24]. Such robots are often designed to be autonomously reconfigurable in which case the robot is capable of disconnecting and reconnecting its modules in different locations.

Although not all such systems are designed as self-replicating systems there is a sense in which many of them are. For example, in an environment of individual disconnected modules, a small 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, Mytilinaios, Adams and Lipson [74] make this explicit by building a modular robotic system in which a configuration of four modules can construct a replica configura-

tion when provided with a supply of additional modules in a location known to the robot. Figure 2.16 shows this system after a replica has been constructed.

Figure 2.16. The self-replicating modular robot of Zykov et al.

A notable feature of the system of Zykov et al. is that a single 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.

2.5.5 The RepRap project

Bowyer et al. [9] have developed a rapid prototyping system based around a 3D printer that is capable of being programmed to manufacture arbitrary 3D objects. Bowyer hopes 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.

The first version of the system (called Darwin) is shown in Figure 2.17. Darwin consists of a pair of nozzles D that can be precisely positioned in two dimensions (left, right, back and forth) by the motors labelled E, and a platform A that can move up and down on screw threads B under the control of motor C.

One of the nozzles is fed with a thermoplastic with a melting temperature between

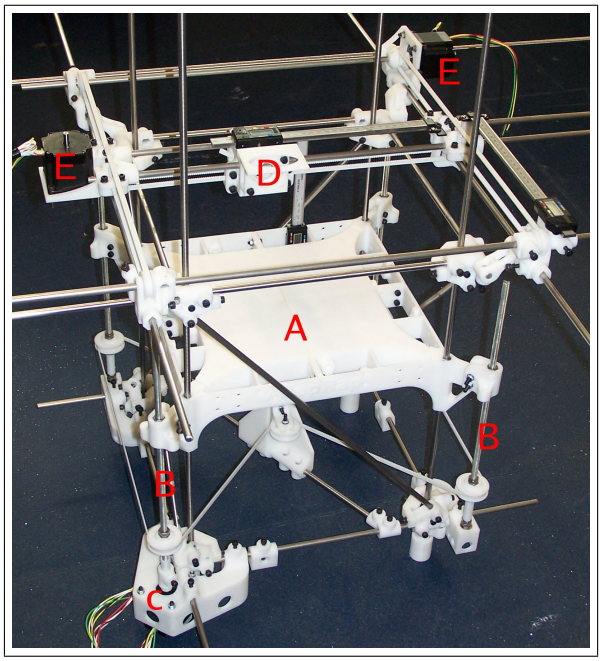


Figure 2.17. RepRap 3D self-replicating printer.

100 and 200 degrees celsius and contains a heater which melts the plastic so that it can be extruded from the nozzle in a 0.5mm diameter stream.

The nozzle and platform move as the plastic is extruded so as to cause the plastic stream to be laid down in any desired shape. For shapes that cannot be made without overhanging portions a second nozzle is used to extrude a supporting medium (for example, icing sugar, which offers good support but which can be dissolved away in water afterwards).

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.

2.6 Proposed and conceptual self-replicating machines with a large constructional capability

2.6.1 NASA study

Self-replicating systems offer enormous promise for the exploration of space. Indeed without exploiting an artificial 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 [22] 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.

The study report also includes 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 [23].

At around the same time as this study was completed, Freitas also proposed a self-replicating interstellar probe [21]. The probe, propelled by a nuclear fusion rocket engine, could be launched towards another star and upon arriving would intelligently assess the environment around the star. Then after transmitting large amounts of scientifically interesting information about the star system back to Earth, it would seek out planetary bodies from which raw materials could be obtained, so that replica probes could be constructed and launched to further stars.

2.6.2 Drexler's assembler

In Engines of Creation [17], 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 constructing 'head' 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.

There may be some serious risks involving nanoscale self-replicating assemblers [8]. Assemblers capable of operating in an outdoors environment and capable of using natural resources as raw materials could malfunction and spread like highly efficient bacteria, consuming all available natural resources on the planet — the so-called 'grey goo' scenario. As a way of mitigating some of these risks, later proposals for molecular manufacturing systems have some element that would otherwise be required for a system to be able to self-replicate in a fully autonomous way provided by an external agent [51]. For example the instructions required for an assembler to be able to construct a replica of itself could be broadcast to every assembler from an external source, which could have a limited range and which could be shut off in the event of a significant malfunction.

There has been some debate about whether molecular manufacturing systems of the kind envisioned by Drexler are possible. Drexler's ideas in 'Engines of Creation' [17] and in his later work 'Nanosystems' [19] are in the form of *theoretical applied science* and therefore allow for a broad range of possible implementations of the concepts described.

Most criticism of Drexler's proposal has focused on the part of the system which transfers atoms or small molecules one-by-one from one location (a storage location) to the structure being assembled.

Smalley [56, 57] argued that molecular assemblers are simply impossible because of incorrect assumptions about the nature of chemistry implicit in the idea of a molecular assembler and that only liquid-phase systems involving large numbers of highly specific enzymes (i.e. systems very much like living organisms) are capable of the synthetic diversity and construction closure that are necessary for molecular-level assembly. Smalley's arguments are not quantitative and do not discuss any particulars of the research that has already been done on the type of systems that he is arguing are impossible. Merkle [42] has carried out in-depth studies on a set of reaction mechanisms that could be used in a first-generation molecular manufacturing system. This research has not yet come up against any insurmountable barriers and unless it does it is premature to say that its aims are impossible or that the potential benefits and dangers that it poses can be safely ignored.

2.7 Simulating self-replicating machines

This review shows that while considerable work has been done on simulating cellular automaton based models of self-replication 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.

The self-replicating systems of von Neumann, Langton, Penrose, Moses, Griffith, Zykov et al. and Bowyer all involve 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 Figure 2.18 in which several of these self-replicating systems are positioned relative to other systems in such a way as to show how they differ from each other in the simplicity of their parts, the range of things that they can construct, and how physically realistic they are.

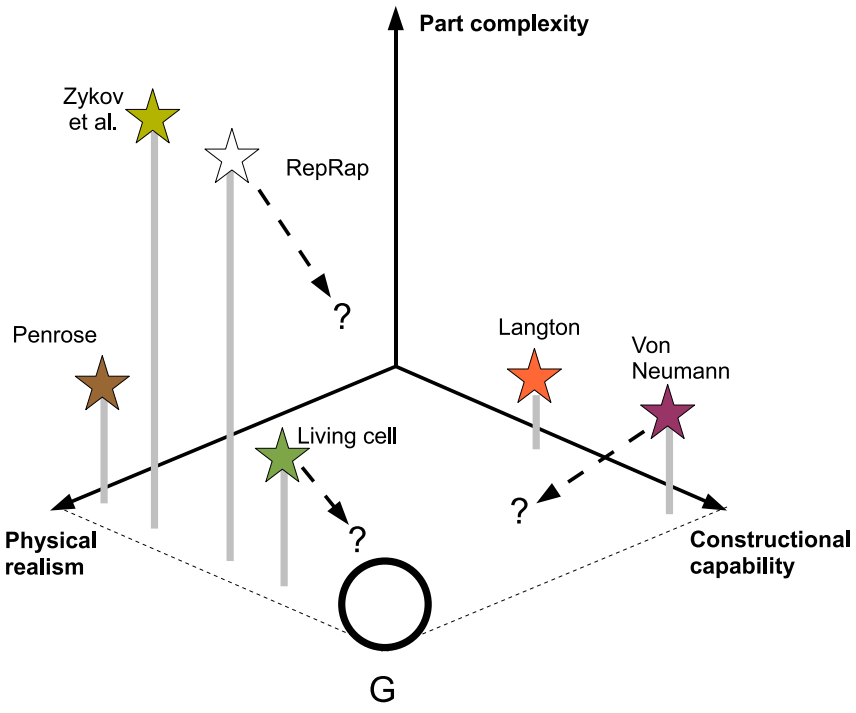


Figure 2.18. Map of some existing work on self-replicating systems.

The region labelled G in Figure 2.18 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. Engineering living cells so as to increase the range of things that they can make is one approach. Another is to begin with a physical system such as the Darwin system used in the RepRap

project, and then work out how to increase the range of things 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 thesis is to investigate the largely unexplored middle ground between simulations of cellular-automaton-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 constructible 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 why it cannot be reached.

Whether modelling or building physical automata, it is necessary to choose an appropriate level of abstraction. What is considered appropriate will depend upon which features of an automaton are of interest and which questions are being asked. For example, if one is concerned with the inputs and outputs of an information processing automaton but not at all concerned with the medium on which the information is written or the speed with which it is processed, the automaton can be modelled at a purely logical level of abstraction. A second example is that of building a digital electronic computing machine. The machine can be built from standard digital circuit elements that are designed so as to behave in such a way that as far as possible a computer designer need not be concerned with what goes on inside the elements but only with the topology and timing of the interconnected parts.

The cellular automata systems that have been used for investigating self-replicating programmable constructors explicitly model logic, time and topology. This research explores self-replicating programmable constructors in simulation environments that in addition model geometry, motion, forces and mechanics. Individual parts in these environments have the ability to move around and a facility is provided by which parts may be connected to one another. 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 named 'kinematic automata'. (A term first used by Burks in [11]). Von Neumann's idea for a kinematic model of a self-replicating system, described in section 2.5.2, inspired much of the work in this thesis.

Chapter 3

Exploratory Work

Chapter 2 shows that existing research into self-replicating programmable constructing systems has focused either on highly abstract simulations, or on physical systems limited by having a high component part complexity or a small constructional capability.

Two pieces of exploratory work were undertaken in the relatively unexplored territory between abstract systems and physical systems in order to evaluate where further research efforts could best be concentrated.

Sections 3.1 and 3.2 summarise these two pieces of work, which have been published in [61] and [60] respectively. Section 3.3 evaluates both and then outlines the direction that the rest of this thesis will follow.

3.1 A 2D discrete space kinematic environment

3.1.1 Introduction

CBlocks is the name of an environment in which square tile-like parts move and interact with one another in a two-dimensional discrete space.

There are several different types of part. Each type performs a specific function. Some types of part perform arithmetical and logical functions, others move in response to a signal or move other parts, and others connect or disconnect other parts. Parts can send and receive integer valued signals to and from neighbouring parts.

Parts can move one unit in any of four directions in a single unit of time. When a part moves into a neighbouring cell that is already occupied, the occupying part gets pushed away. A set of rules determine how signals pass between parts, how parts can be connected together and how they should behave when they are connected. Parts can be in any one of four possible orientations.

3.1.2 Detailed description of CBlocks

We define four direction vectors

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

Let D denote the set of these vectors

$$D = \{NORTH, EAST, SOUTH, WEST\}$$

We define the function

$$\text{opposite}((x,y)) = (-x,-y)$$

Let L be the set $\{True, False\}$, and let T denote the set of part types

$$\begin{aligned} T = \{ \quad & Wire, Cross, Delta, Not, And, Or, Nand, Nor, Insulator, \\ & Push, Thrust, RFuse, LFuse, RUnFuse, LUnFuse, RSlide, LSlide, \\ & Equal, Pulse, Creator, Multiplier, Adder, Store, Toggle\} \end{aligned}$$

A part P is completely described by the tuple

$$(P.location, P.orientation, P.type, P.output, P.connect)$$

where

$$\begin{aligned} P.location &\in \mathbb{Z} \times \mathbb{Z} \\ P.orientation &\in D \\ P.type &\in T \\ P.output &\in \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \\ P.connect &\in L \times L \times L \times L \end{aligned}$$

P.location is a tuple (x, y) which specifies the location of the part in space. *P.orientation* is the orientation of P.

The notation $X[Y]$ is used to refer to the Yth 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 $NORTH, EAST, SOUTH$ and $WEST$ can be used to index the 1st, 2nd, 3rd and 4th elements of a tuple respectively.

$P.output[d] \in \mathbb{Z}$ where $d \in D$ are the outputs of P. So for example if we have an isolated *Nor* part P with $P.orientation = EAST$, the values of its outputs will be $P.output[EAST] = 1$ and $P.output[d] = 0$ 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 $P.connect[d] = False$.

The edges of parts can be regarded as terminals through which signals can be passed between neighbouring parts. 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 signal with a value of zero.

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 3.1 describes 24 part types, 23 of which are used in section 3.1.3. (The *RUn-Fuse* part is not used, but is included in Table 3.1 for completeness). In Table 3.1 the letters N,S,E and W are used to refer to the values of the input signals at the $NORTH, SOUTH, EAST$ and $WEST$ edges of a part. These letters are also used to indicate directions, where N is up the page, E is to the right of the page, S is down the page and W is to the left of the page. The context should indicate which usage is meant.

An abbreviated notation is used for describing the relationship between a part's inputs and outputs. So, for example, if a part P lies in direction S of a *wire* part Q then when referring to part Q the abbreviation $N := S$ is used to mean that after one time unit, $Q.output[N]$ will take on the value of $P.output[N]$. In the abbreviated notation $N := S$ the N on the left hand side refers to the $North$ terminal of Q, and the S on the right hand side refers to the direction to part P: the direction that the input will be received from.

1 Wire  N:=S	2 Cross  N:=S,E:=W	3 Delta  N,E,W:=S
4 Not  N:=!S	5 And  N:=min(E,W)	6 Or  N:=max(E,W)
7 Nand  N:=!(E&&W)	8 Nor  N:=!(E||W)	9 Insulator 
10 Push  When S!=0, push on part that lies N, in the N direction	12 Thrust  When S!=0, push on self in the S direction	13 RFuse  When S!=0, connect the parts that lie N and NE
14 LFuse  When S!=0, connect the parts that lie N and NW	15 RUnFuse  When S!=0, disconnect the parts that lie N and NE	16 LUnFuse  When S!=0, disconnect the parts that lie N and NW
19 RSlide  When S!=0, apply a force on part that lies N, in the E direction	20 LSlide  When S!=0, apply a force on part that lies N, in the W direction	21 Equal  N:=(E==W)
22 Pulse  N:=S when S changes from 0 to non-zero N:=0 otherwise	24 Creator  Create a part in the N direction, with type given by S	25 Multiplier  N:=E*W
26 Adder  N:=E+W	28 Store  If S!=0 and output N==0, set output N to S. If E!=0 or W!=0, set output N to 0	32 Toggle  If S!=0, toggle the value of output N

Table 3.1. Part types supported by the CBlocks environment.

The notation used for expressions in Table 3.1 is that used by the C programming language, summarized in Table 3.2.

Operator	Name and meaning
+	Addition Sum of operands
*	Multiplication Product of operands
==	Equal to 1 if operands are equal, zero otherwise
!=	Not equal to zero if operands are equal, 1 otherwise
!	Logical NOT 1 if operand is zero, zero otherwise
&&	Logical AND 1 if both operands are non-zero, zero otherwise
||	Logical OR 1 if any operand is non-zero, zero otherwise

Table 3.2. Operators used in Table 3.1.

3.1.3 A self-replicating programmable constructor in the CBlocks environment.

A self-replicating programmable constructor (SRPC) has been devised in the CBlocks environment using the part types described in the previous section. The SRPC would be far more complex were it not for the *creator* part type. This part type allows new parts to be created from nothing in response to an integer signal that encodes the type and orientation of the part to be created in a way that will be explained shortly.

Figure 3.1 illustrates the geometrical structure of the SRPC; the four main subsystems are labelled. The *instruction hopper* contains a block of *store* parts which encode a sequence that directs the SRPC to move around in its environment and create parts in such a way as to construct another machine. This sequence of *store* parts is referred to as the *instruction tape*. The length of the *instruction tape* places a limit on the size of machine that the SRPC can construct. Self-replication is achieved when the *instruction tape* contains a sequence of instructions for constructing a duplicate machine.

In the description of the SRPC that follows the terms *parent* and *child* are used to refer to machines in a relationship where one instance of the machine has constructed or is constructing another.

Figure 3.2 shows how the *instruction tape* is arranged in the machine. The arrows

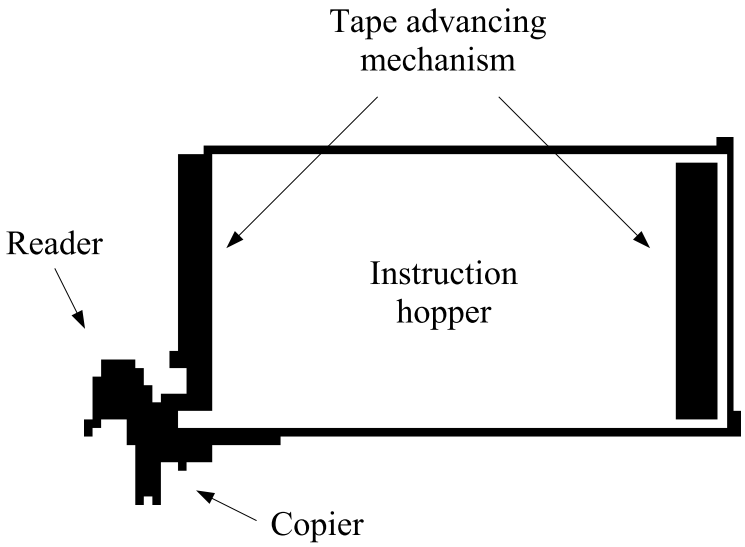


Figure 3.1. The geometrical structure of the SRPC.

show the direction in which *store* parts move as the tape is advanced. The *tape-advancing mechanism* ensures that the tape advances one part at a time and that the arrangement shown in Figure 3.2 is maintained.

Figure 3.2. The arrangement of the *instruction tape*.

Figure 3.3 shows successive steps in the operation of the *tape advancing mechanism*. In Figure 3.3 the size of the tape has been reduced and a characteristic portion of the mechanism is shown. Blue lines between parts in Figure 3.3 represent connections between parts. When two neighbouring parts are not connected they have a black line between them. In step A the mechanism is not active. In step B signals enter *delta* parts 1 and 2 on the left-hand and right-hand sides of the mechanism. The signal on the right-hand side of the mechanism enters *rslide* part 3 so that in step C *rslide* part 3 slides a *store*

part out of the way, leaving gap 4. Next *push* part 5 is activated, pushing the top row of three *store* parts to the right, resulting in gap 6 as shown in D. After this *lslide* part 7 is activated, causing *store* part 8 to move into gap 6. Then *push* part 9 pushes the bottom row of three *store* parts to the left, resulting in gap 10 as shown in F.

It should be straightforward to see how the portion of the mechanism shown in Figure 3.3 leads to the tape-advancing motion depicted in Figure 3.2.

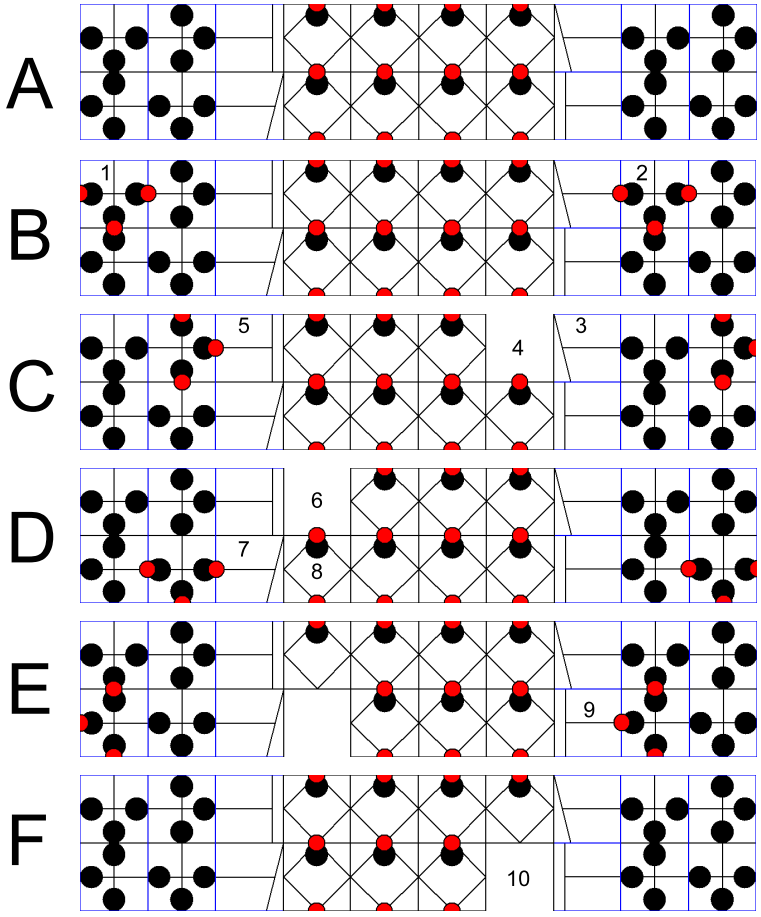


Figure 3.3. The operation of the *tape advancing mechanism*.

The *reader* contains logic that interprets signals from the instruction tape and acts upon them. Figure 3.4 shows the logical structure of the reader.

The SRPC uses a *creator* part to create new parts as they are needed. This part creates a new part whose type and orientation (i.e. orientation relative to the *creator* part in the *reader*) are dependent on the value of the input signal that it receives. A signal value of $4T + D$ encodes a part of type T and orientation D. Values of T for each part

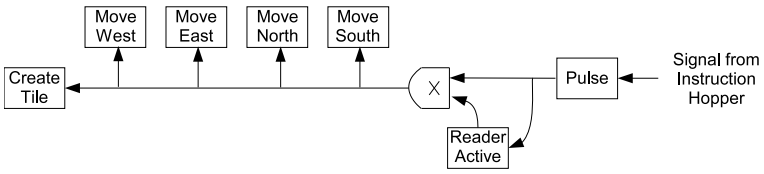


Figure 3.4. The logical structure of the *reader*.

type are given in Table 3.1. The mapping between values of D and possible orientations is $\{(0, N), (1, E), (2, S), (3, W)\}$.

Some of the *store* parts in the *instruction tape* encode values which tell the SRPC to perform an action. The values used are as follows:

$1001 =$ move east

$1002 =$ move south

$1003 =$ move west

$1004 =$ move north

$1005 =$ switch between *read* and *copy* phases

$1006 =$ do nothing

No explicit instruction is needed in order to tell the *reader* to fuse newly created parts together since the *reader* contains *fuser* parts that are always active and which fuse together any parts that pass in front of them.

The *copier* is responsible for creating a duplicate instruction tape in a child SRPC. Figure 3.5 shows the logical structure of the *copier*.

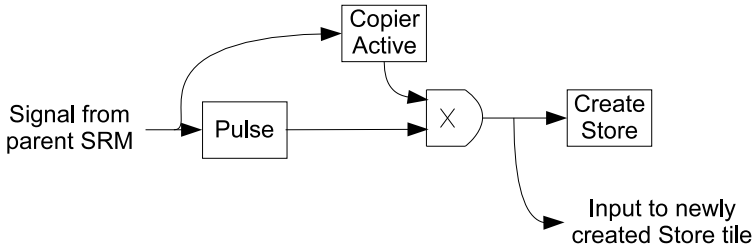


Figure 3.5. The logical structure of the *copier*.

The replication cycle has two phases: the 'reading phase' and the 'copying phase'. During the reading phase the *instruction tape* is interpreted by the *reader*. The last

instruction in the instruction sequence (code 1005) causes the machine to switch between the reading and copying phases. During the copying phase the parent sends signals to the child which cause a copy of the *instruction tape* to be created in the child SRPC. The child machine is then complete and can begin constructing its own child. The parent machine switches back to the reading phase and the replication cycle begins again. Figures 3.6 and 3.7 show two snapshots of the SRPC in action, showing which phase it is in and what it is doing at each snapshot.

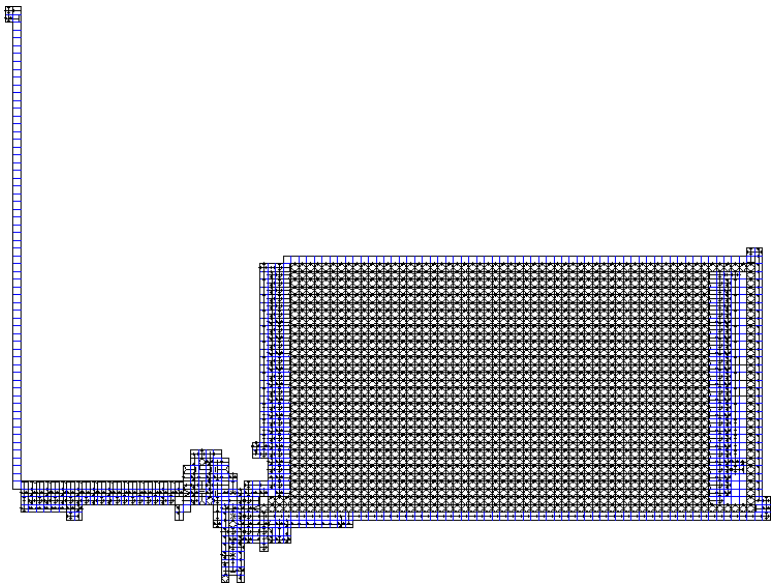


Figure 3.6. The parent SRPC in the reading phase, part way through constructing a child SRPC.

Figure 3.8 shows a parent SRPC and the child SRPC that it has produced. Notice that the child is constructed so as to be oriented 90 degrees anticlockwise with respect to the parent. This is done so that successive generations of SRPCs will fill up the two-dimensional universe. It might be argued that because of this difference in orientation the child is not an exact replica of the parent. However, since the CBlocks environment is rotationally symmetric the logical and kinematical behaviour of parent and child can reasonably be regarded as equivalent.

Figure 3.9 shows the state of the universe after the initial SRPC has produced two child SRPCs, the first of which has produced a child of its own.

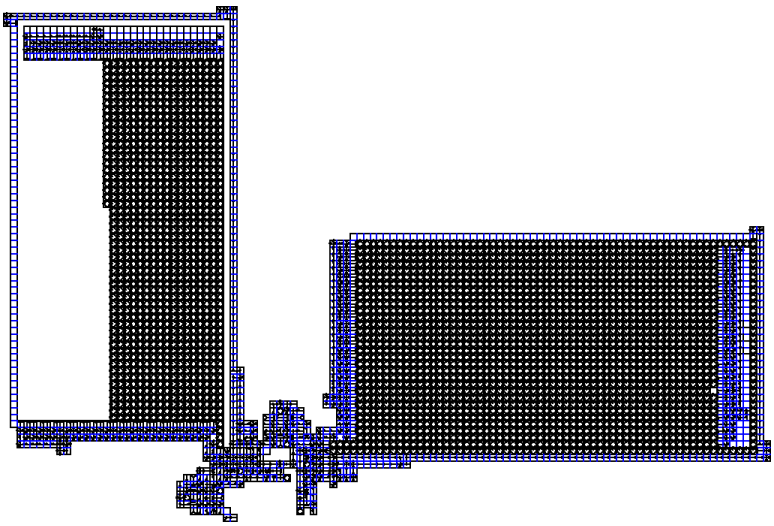


Figure 3.7. The parent SRPC part way through the copying phase.

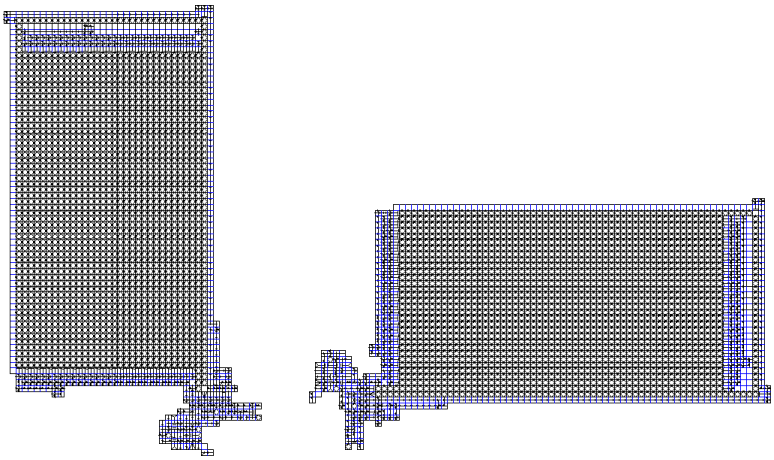


Figure 3.8. A parent SRPC has produced a child.

3.1.4 Physical realism in the CBlocks environment

In what sense is the CBlocks environment more physically realistic than a cellular automaton? Cellular automata can of course be made from arrays of discrete parts with each physical part corresponding directly to a cell in the abstract environment, as in [7]. A self-replicating system in such an environment would be able to alter the internal state of a discrete part in the array but this would be the limit of its effect on the physical environment. Such a system does not harness the mechanics of the physical environment for the purpose of self-replication.

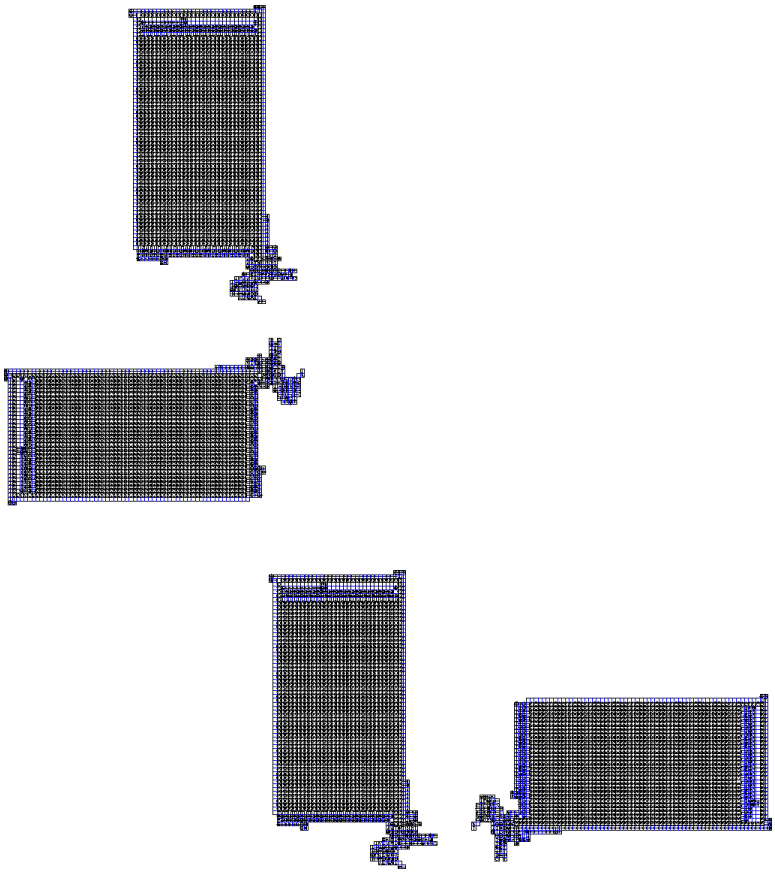


Figure 3.9. Three generations of SRPCs.

A physical self-replicating system capable of building itself from component parts would have to make use of the mechanics of the environment in which the component parts function in order to replicate itself. The CBlocks SRPC does this within the CBlocks environment. This environment has rules of motion and interaction loosely based upon the laws of motion and the mechanical interactions of physical machines. In this sense the CBlocks environment can be said to be more physically realistic than cellular automata.

However, matter is not conserved in the CBlocks environment, since a *creator* part can create other parts from nowhere. At first sight it seems that a *creator* part is not physically realistic. Nevertheless, it is possible to envisage physical systems in which something like a *creator* part can be made. For example, if a physical model based on CBlocks existed on a two-dimensional surface and a second surface were placed just above this surface then the second surface could contain a disorganised collection of parts moving about at random. A *creator* part on the lower surface needing to create a part could wait for a part

of the correct type and in the correct orientation to pass above it on the upper surface and then cause it to be transferred from the upper to the lower surface.

3.2 A 2D continuous space kinematic environment

3.2.1 Introduction

Having shown that it is possible to devise an SRPC in the CBlocks environment consideration is now given to the implementation of a self-replicating system in an environment supporting the same types of part as the CBlocks environment, but in which space is continuous and parts interact according to Newtonian laws of motion.

Nodes is an environment in which disc-shaped parts interact with each other in a boundless two-dimensional continuous space environment. There are several different types of part, with each type of part performing a specific function. There are parts that perform logical functions, parts that connect other parts together and parts that exert forces and cause motion. Parts have four terminals which are used to send and receive integer valued signals to and from other parts. The motion of parts is governed by Newtonian laws. In addition to these laws there are also rules that determine how signals pass between parts, how parts can be connected together and how they should behave when they are connected. Automata can be constructed from collections of parts connected up in an appropriate way.

3.2.2 Detailed description of the *Nodes* environment

In Nodes time is discrete and moves forward in steps of 1 unit.

A part L has the fixed property $L.type$ which specifies the type of L and does not change from one time step to another. L also has the property $L.identifier$, which is a positive integer unique to each individual part. ($L.identifier$ is used when two parts are connected together and each part needs to keep track of which parts it is directly connected to).

Other properties of L can vary from one time step to another. These properties are:

$L.location \in \mathbb{R} \times \mathbb{R}$ is the location of L in two dimensional space. The notation $\overrightarrow{LM}$ is short for $M.location - L.location$. $L.location.x$ refers to the x coordinate of L and $L.location.y$ refers to the y coordinate of L.

$L.orientation \in [0, 2\pi)$ is the orientation of L in radians anticlockwise from the vector (0,1).

$L.velocity$ is the velocity of L. $L.angularvelocity$ is the angular velocity of L.

Every part has a mass of 1 unit and a moment of inertia about its centre of 1 unit. Therefore momentum is numerically equal to velocity, and the angular momentum and angular velocity of a part about its centre are also numerically equal.

Forces that act on a part arise in several different ways. All parts in the universe experience a frictional force proportional to their velocity, and an angular frictional force proportional to their angular velocity.

$friction(L) = c \times L.velocity$
$angularfriction(L) = d \times L.angularvelocity$

Parts have a radius of 8 unit and there is a repulsive force between any pair of overlapping parts L and M.

If $\|\overrightarrow{LM}\| < 16$ and $\|\overrightarrow{LM}\| > 0$ then $repulsion(L, M) = f \times \frac{\overrightarrow{LM}}{\|\overrightarrow{LM}\|} \times (16 - \|\overrightarrow{LM}\|)^2$
otherwise, $repulsion(L, M) = 0$

Parts can be connected to other parts using four terminals that are evenly spaced around the edge of a part.

$L.connect[N], L.connect[E], L.connect[S], L.connect[W]$ specify the connectivity state of a part L. If a terminal t of a part P has no connection then $P.connect[t] = 0$. If a part P is connected via a terminal t to a part L then $P.connect[t] = L.identifier$ and there will exist a terminal s of L such that $L.connect[s] = P.identifier$.

When two parts are connected they exert forces on each other to bring themselves into proximity. The function $rotate(v, a)$ rotates the vector v through angle a.

$proximity(P, L) = g \times (P.location + 2 \times rotate(t, P.orientation) - L.location)$

Angular forces are also exerted on connected parts, firstly to direct the terminal t to point towards the connected part L:

$$direction(P, L) = h \times angle(\overrightarrow{PL}, rotate(t, L.orientation))$$

where $angle(A, B) \in [-\pi, \pi)$ is the angular difference between vectors A and B.

Secondly to ensure that terminal t is pointing in the opposite direction to terminal s:

$$alignment(P, L) = j \times angle(-rotate(t, P.orientation), rotate(s, L.orientation))$$

Let L_t denote the state of part L at time t.

We now have enough information to define for a universe full of parts how L_{t+1} depends upon L_t and all parts M that L interacts with.

$$\begin{aligned}
L_{t+1}.location &= L_t.location + L_t.velocity \\
L_{t+1}.orientation &= L_t.orientation + L_t.angularvelocity \\
L_{t+1}.velocity &= L_t.velocity + friction(L) + \\
&\quad \textstyle\sum_M (repulsion(L, M) + proximity(L, M)) + E \\
L_{t+1}.angularvelocity &= L_t.angularvelocity + angularfriction(L) + \\
&\quad \textstyle\sum_M (direction(L, M) + alignment(L, M))
\end{aligned}$$

where E represents any possible forces applied on a part L by one of the part types *RSlide*, *LSlide*, *Push* or *Thrust*.

The system as described is capable of exhibiting a wide range of behaviours depending on the choice of values for the constants c, d, f, g, h and j. For example, by having no friction we obtain a system that behaves like an ideal gas in which particles collide and rebound off one another endlessly. The following considerations were taken into account when choosing values for these constants:

- Given that time is discrete the maximum speed that a part can reach should not be so large that a part can move a significant fraction of its diameter in a single time unit. (The maximum speed depends both on the friction constants and the applied force constant).

- The forces between connected parts should be set so that the level of damping caused by the friction forces is neither so great that connected parts take a long time to come into alignment nor so small that connected parts oscillate excessively.

- With a view to possible future physical realisation of the system the friction constants should be chosen so that parts move somewhat like discs floating on the surface of water.

- Constants should be chosen so that the time taken to run simulations is not excessive.

The values: $c = 0.9$, $d = 0.9$, $f = 0.1$, $g = 0.04$, $h = 0.05$, $j = 0.1$ were found to satisfy these considerations. These values also result in the motion of parts being generally much slower than the propagation of signals between parts.

Note that the system is deliberately defined as a discrete time system rather than as an approximation to a continuous time system so that the system described above can be simulated without any ambiguity as to whether the results of simulating the system are dependent on the degree of approximation used.

3.2.3 Signals

The terminals which are used to connect parts together are also used as inputs and outputs for passing signals between parts. Parts do not need to be connected in order for signals to pass between them. If an output terminal is within a distance of 0.5 units from an input terminal then any signal it is outputting will be received by the input terminal. Each terminal is either an input or an output. If a terminal has no explicit definition it is effectively an output producing a signal value of zero. The absence of a signal corresponds to a value of zero.

$L.output[N], L.output[E], L.output[S], L.output[W] \in \mathbb{Z}$ are the outputs of L.

Note that unlike in CBlocks, and because the orientation is a continuous value, the direction of terminals are specified relative to the orientation of a part so, for example, if we have an isolated *Not* part L the values of its outputs will be $L.output[E] = L.output[S] = L.output[W] = 0, L.output[N] = 1$ regardless of its orientation.

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. A part's type determines how it responds to input signals and whether it produces any output signals.

The *Push*, *RSlide* and *LSlide* parts described in Table 3.3 exert forces on other parts when activated by a signal. The *Thrust* part exerts a force on itself when activated by a signal. For each of these parts, the magnitude of the force is the constant $k = 0.5$.

3.2.4 Part types

Table 3.3 describes 23 part types. In Table 3.3 the letters N,S,E and W (for North, South, East and West) are used to refer to terminals and also to indicate directions. The context should indicate which usage is meant.

The notation used for expressions in Table 3.3 is that used by the C programming language, summarized in Table 3.2.

3.2.5 A self-replicating machine in the Nodes environment

The kinematics of the Nodes environment are much more complex than those of the CBlocks environment and this makes it much more difficult to devise easily-controllable mechanisms in the Nodes environment.

For example whereas in the CBlocks environment the behaviour of a rigid structure under the influence of, say, a *RSlide* part is easy to state, the same is not true in the Nodes environment where the behaviour of a connected set of parts influenced by a force produced by a *RSlide* part depends on the number of parts in the set (i.e. the mass of the set) and the position of the point of application of the force relative to the centre of mass of the set of parts.

Also the computational requirements for simulating the Nodes environment are significantly more than for the CBlocks environment, largely because of the more complex laws of motion.

Therefore, the first question to be asked about the Nodes environment, using the same part set as in the CBlocks environment, is whether it is capable of supporting any type of self-replicating system *at all*, let alone one capable of being programmed to construct other structures. The following sections show that it is indeed possible to devise a self-replicating system in the Nodes environment, albeit one with a limited constructional capability.

3.2.6 Filaments and self-assembly

A linear chain of parts is referred to as a filament. An experimental investigation into the behaviour of parts and structures in the Nodes environment revealed that it is straight-

1 Insulator 	2 Wire  N:=S	9 Delta  N,E,W:=S
3 Not  N:=!S	10 NDelta  N,E,W:=!S	15 Cross  N:=S, E:=W
22 NotNot  N:=!!S	12 Or  N:=(E||W)	21 Maj  N:=E*W+E*S+S*W
17 Pulse  N:=S only when S changes from zero to non-zero. N:=0 otherwise	8 FlipFlop  When E!=0, set N and S to 1. When W!=0 set N and S to 0.	24 Equal  N=E&&(E==W)|| E&&(E==S)|| S&&(S==W)
0 Store  N set to S when S changes from zero to non-zero, reset when E or W!=0	7 LFuse  When S!=0 connect the parts that lie N and NW	6 RFuse  When S!=0 connect the parts that lie N and NE
14 LUnFuse  When S!=0 disconnect parts that lie N and NW	13 RUnFuse  When S!=0 disconnect parts that lie N and NE	11 Detect  S is non-zero only when a part lies N
4 Thrust  When S!=0 apply a force on self in S direction	23 Push  When S!=0 apply a force on part that lies N in the N direction	19 LSlide  When S!=0 apply a force on part that lies N in the W direction
18 RSlide  When S!=0, apply a force on part that lies N in the E direction	20 Creator  Create a part in the N direction, with type given by S	

Table 3.3. Part types in the Nodes environment.

forward to devise a machine M that will advance along a filament. If the filament is made from *store* parts then signals from these *store* parts can be fed to a *creator* part in M which will create other parts with type and orientation dependent on the values stored in the *store* parts. A *fuse* part in M can be placed near to the *creator* part in such a way as to connect the newly created parts together. A value of 1 in a *store* part will not cause any part to be created but will instead leave a gap in the sequence of created parts. The filament of *store* parts can thus be programmed to cause any desired series of filaments to be created (Figure 3.11).

By investigating how filaments containing *slide*, *thrust* and *push* parts behave, it was found possible to arrange for pairs of filaments to assemble to form structures 2 parts thick. For example, the pair of filaments in Figure 3.11 will move as shown in Figure 3.12 to form a 16 by 2 part structure.

3.2.7 Subsystems of the self-replicating machine

The self-replicating machine (SRM) that was devised in the Nodes environment is made from several such 2-part-thick structures, each of which carries out a particular function in the machine. The machine is not a single connected collection of parts as the CBlocks SRPC was, but is a disconnected collection of smaller machines, each referred to by a descriptive name. These machines are introduced below, along with acronyms that are used to refer to them later on. Some of the acronyms are used before they are defined.

- Instruction Tape (IT): a filament of store parts that contains the information needed for the machine to make a duplicate of itself.

- Tape Copier (TC): a machine that copies the Instruction Tape and its contents to make a duplicate Instruction Tape (IT2).

- Dragger (D): a machine that drags IT2 and EFTR to a new location in the environment.

- Releaser (R): a machine that moves along IT2, slowly catching up with D. When R has caught up with D, it causes D to release IT2.

- Initial Tape Reader (ITR): a machine that creates filaments of parts according to information in IT. This machine begins the first replication cycle. The filaments of parts created self-assemble to form TC, RTR, EFTR, R and D.

- End-Finding Tape Reader (EFTR): a machine that attaches itself to the end of IT2 as it is being dragged away, and which begins a new replication cycle when IT2 has reached its destination by performing the same function as ITR.

- Rotating Tape Reader (RTR): a machine that slowly rotates so as to encounter IT after all other activities have stopped, and then performs the same action as ITR, except that it omits the construction of TC, so that the instances of EFTR, R and D that RTR creates will act on IT, rather than a duplicate of IT, ultimately causing IT to be dragged to a different location in the environment and another replication cycle to begin.

As was the case in the CBlocks environment, the SRM would be far more complex were it not for the *creator* part type which is the least physically realistic part in the Nodes environment. The *creator* part will create a part of type T (from Table 3.3) and orientation $D\pi/2$ when fed a value of $1 + 4T + D$ where $D \in \{0, 1, 2, 3\}$.

Figure 3.10 shows the machine shortly after it starts. ITR is advancing along IT, creating a filament of parts.

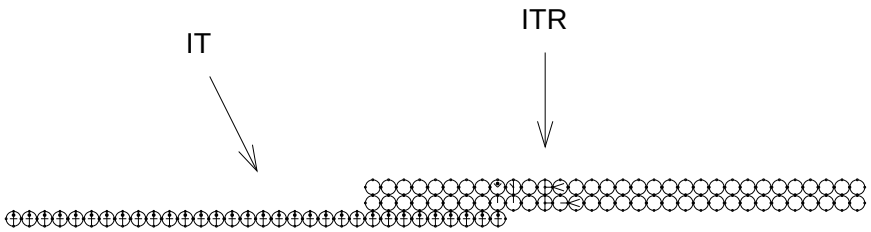


Figure 3.10. The initial configuration of the environment.

Figure 3.11 shows the machine after the two filaments that will assemble to form TC have been created and Figure 3.12 shows how these filaments come together to form TC.

In Figure 3.13 TC, RTR, EFTR, R and D have all been created and TC is beginning to create IT2. In Figure 3.14 D has attached itself to IT2 and is beginning to drag IT2 away.

Figure 3.15 shows EFTR attaching itself to the end of IT2. Also R has been activated and is beginning to move towards D. In Figure 3.16 R has caught up with D, causing D to release IT2.

Figure 3.17 shows EFTR beginning to fold up after IT2 stops moving. Figure 3.18 shows a new replication cycle beginning with IT2 and EFTR.

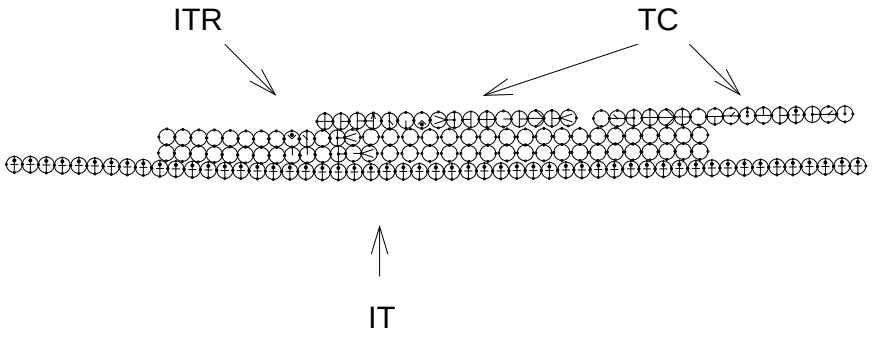


Figure 3.11. The two filaments that form TC have been created.

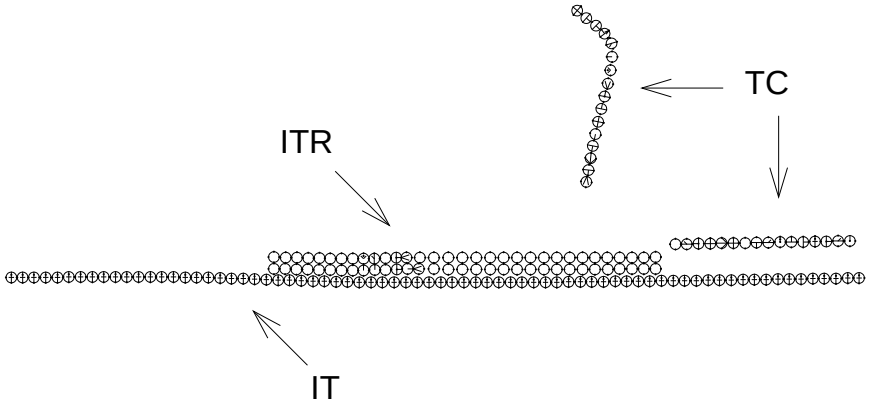


Figure 3.12. TC assembling itself.

Figure 3.19 shows part of the environment after several replication cycles. There are 5 machines in this part of the environment, all of which are still functioning correctly, all descended from a single original machine. Occasionally machines are rendered incapable of further replication by collision with, and interference from, parts of other machines.

3.3 Evaluation and research directions

This chapter has shown that it is relatively straightforward to devise self-replicating systems in kinematic environments supporting a rich set of part types and a special part that can create any other part out of nowhere.

Referring to Figure 2.18 we can argue that we have moved from the abstract side of the graph towards the physically realistic side at the expense of having a moderately large set of parts, at least one of which (the *Creator* part) is highly complex.

The number of each type of part used in the CBlocks and Nodes SRMs is given in

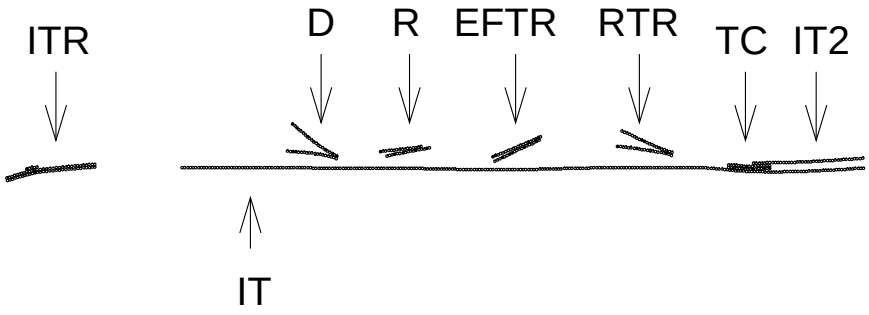


Figure 3.13. TC is beginning to duplicate IT.

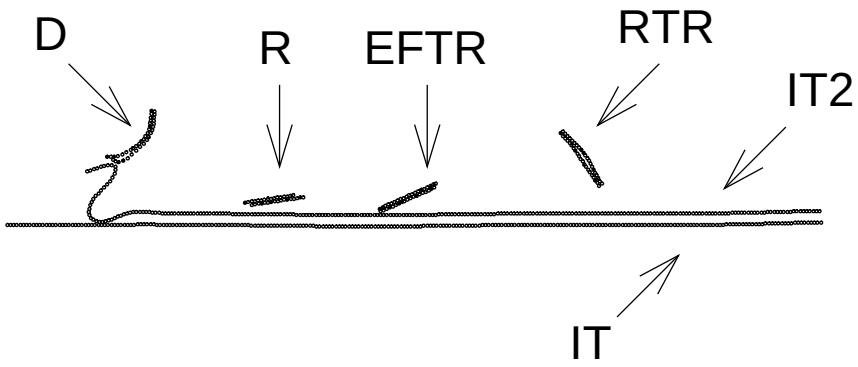


Figure 3.14. D dragging IT2 away from IT.

tables 3.4 and 3.5 respectively. These tables show that in each SRM a handful of parts are used a great deal, whereas other parts are used very infrequently.

There is also considerable redundancy in the set of components used in both environments. In the CBlocks environment, for example, there are 12 different types of part that perform arithmetical or combinational logic functions, and 6 types of part that exert forces on other parts.

Region G in Figure 2.18 on page 36 represents the class of autonomous physical self-replicating systems made from simple parts or raw materials and capable of constructing a wide range of structures. In order to progress from the systems presented in this chapter towards this goal the following three questions need to be addressed.

1. Can we devise an SRPC in a kinematic environment that does not make use of a *Creator* part?

2. Can the range of part types used be reduced to remove the redundancy that is

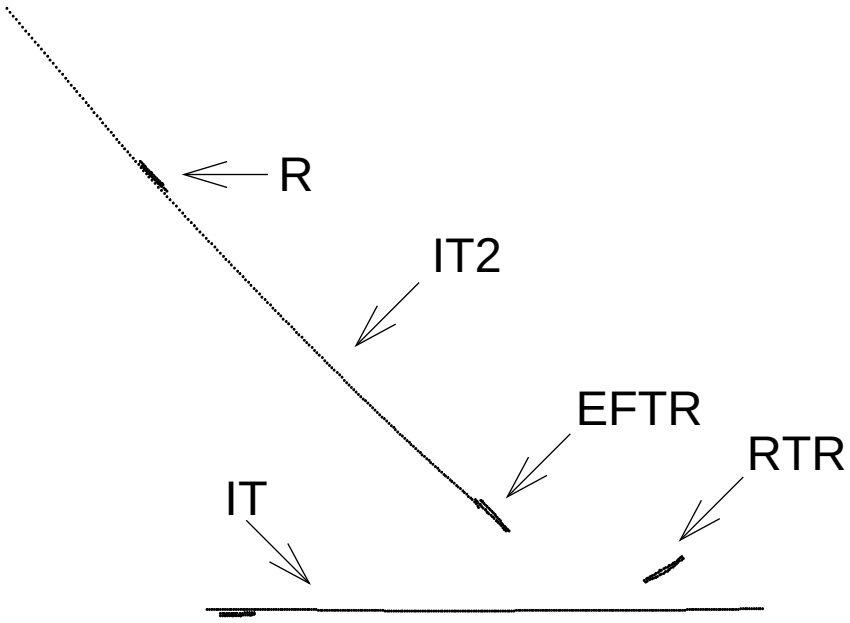


Figure 3.15. EFTR attaching itself to IT2. R is moving towards D

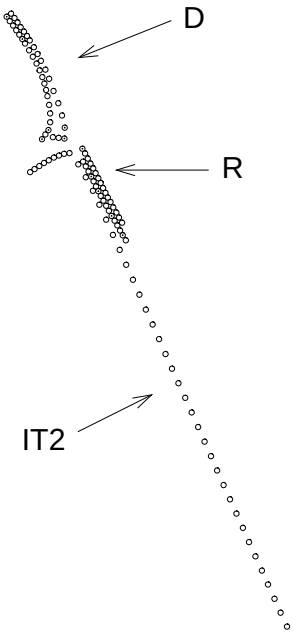


Figure 3.16. R causing D to release IT2.

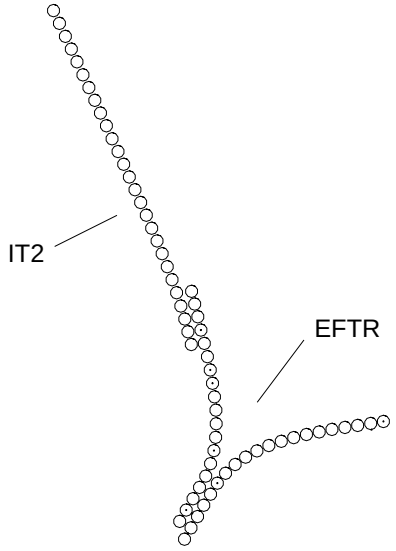


Figure 3.17. EFTR folding up.

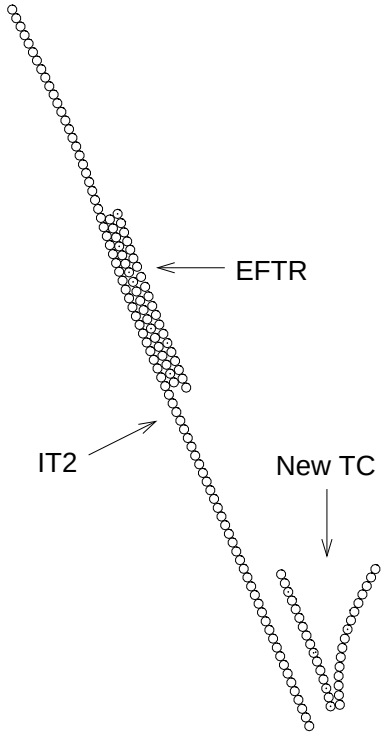


Figure 3.18. EFTR beginning a new replication cycle on IT2.

Figure 3.19. Part of the environment after several replication cycles.

Store	1777 in the instruction tape, 5 elsewhere
Delta	172
Wire	124
Insulator	108
Pusher	32
Left-slider	18
Right-slider	15
Toggle	12
15 Others	48 (less than 6 of each, four part types only used once)

Table 3.4. Frequency of the different types of part used in the CBlocks SRPC.

Store	251 in the instruction tape, 4 elsewhere
Delta	43
NDelta	19
Toggle	18
Rfuse	14
Lfuse	10
Thrust	10
Wire	10
14 Others	129

Table 3.5. Frequency of the different types of part used in the Nodes SRM.

present in the set of part types used in the CBlocks and Nodes environments?

3. Can the environments themselves be made more physically realistic?

In this thesis we focus on questions 1 and 2 and hope to find out whether it is possible in principle to devise an SRPC in a kinematic environment made from parts chosen to be as simple as possible without considering how the parts could be physically implemented.

An SRM without a *Creator* part will need subsystems to carry out the following tasks.

- Fetch parts from a known location or from a disorganised collection.
- If parts are not already categorised then discriminate between different parts in order to classify them.
- Store parts (this is not strictly necessary, but an SRM that does this is likely to be more efficient than one that does not).
- Transport parts to the construction area.

To begin to answer question 2, consider the different roles that individual parts in CBlocks and Nodes fulfill, shown in table 3.6.

Structure
Logic, arithmetic and signal propagation
Exerting forces on parts
Rotating parts
Connecting parts together
Disconnecting parts

Table 3.6. Roles of component parts in the CBlocks and Nodes environments.

Any type of part can perform a structural function, therefore it seems plausible that an SRPC in an environment like CBlocks could be made using only five different types of part. In order to rotate parts in this type of environment, which has laws of motion that do not support the rotation of objects, a special purpose 'rotating' part is required. In the Nodes environment the laws of motion implicitly permit part-rotation, so only four different types of part may be needed to make an SRPC in this environment.

A decision must be made at this point about whether to use a discrete space or a continuous space environment to answer questions 1 and 2 given above. Two disadvantages of the continuous-space Nodes environment have already been discussed in 3.2.5: the computational requirements for simulating the environment are high, and although the

environment is deterministic the detailed motion of structures within the environment is difficult to predict and therefore difficult to reason about. For these reasons continuous space environments are not pursued any further in this thesis. Since we will be using a discrete space environment, and since we envisage that the set of part types that we will be using will be akin to a subset of the set used in the CBlocks environment, we also decide at this point that time shall be discrete.

3.4 Two or three dimensions?

The signal crossing problem is the problem of how two signal-propagating paths can cross over in a 2D plane and how they can do so without interfering with each other's operation [16]. This problem can be solved in a number of ways depending on the signal representation used and the type of component parts that are available for making signal paths. When attempting to make the range of part types as simple as possible signal crossing becomes an inconvenience. The CBlocks environment has a *cross* part that performs a signal crossing function. If we wish to do away with this part and implement it using, for example, a collection of NOR-gates then there is no difficulty in doing this but 12 gates are required.

In a two dimensional kinematic environment signal crossing mechanisms based on moving parts can be made. The operation of the mechanism will depend upon the part types that the environment supports. A simple example is given in Figure 6.3 on page 148.

By moving from two dimensions to three dimensions we can remove the signal crossing problem altogether.

Another benefit that 3D space has over 2D space is that the extra dimension permits a larger number of mechanisms to be placed within a given distance of a particular location in space. This reduces the time that it takes for information to propagate from one mechanism to another, potentially increasing the operating speed of a collection of mechanisms.

Having six sides from which to access a part rather than four is also beneficial. It reduces signal routing congestion problems and also makes it easier for a part to be both examined and manipulated within the same mechanism, which has relevance for question 1.

In the following chapters these considerations are taken into account and questions 1 and 2 are both answered.

Chapter 4

A 3D Kinematic Environment with 6 Part Types

4.1 Introduction

Following the considerations of section 3.3 this chapter describes a discrete space, discrete time three-dimensional kinematic simulation environment called CBlocks3D. This is an extension into three dimensions of the two-dimensional CBlocks environment described in section 3.1. The range of part types in CBlocks3D has been reduced considerably from that in CBlocks. The signals that pass between parts are Boolean valued in CBlocks3D rather than integer valued as in CBlocks. There are only 6 different part types: 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). 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.

This chapter describes the environment, its rules of motion and the parts that it supports, and then describes some of the simple structures that can be made in this environment. Chapter 5 then describes a self-replicating programmable constructor with subsystems made from or based upon the simple structures described in this chapter.

4.2 Describing parts

We define six direction vectors

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

Let D denote the set of these vectors

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

We define the function

$$\text{opposite}((x,y,z)) = (-x,-y,-z)$$

Let L be the set $\{True, False\}$, and let T denote the set of part types

$$T = \{Wire, Nor, Slide, Fuse, UnFuse, Rotate\}$$

A part is completely described by the tuple

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

where

$$\begin{array}{rcl} P.location & \in & \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \\ P.primary & \in & D \\ P.secondary & \in & D \setminus \{primary, \text{opposite}(primary)\} \\ P.type & \in & T \\ P.output & \in & L \times L \times L \times L \times L \times L \\ P.connect & \in & L \times L \times L \times L \times L \times L \end{array}$$

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 three dimensional 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 4.1 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 Yth 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 $Q.connect[\text{opposite}(d)] = True$. If a part P is not connected to its neighbour Q which lies in direction d, then $P.connect[d] = Q.connect[\text{opposite}(d)] = False$. If a part P has no neighbour in direction d then $P.connect[d] = False$.

The faces of parts can be regarded as terminals through which Boolean signals can be passed between neighbouring parts. 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 4.1 describes the function of each type of part. A graphical representation and colour coding scheme is also shown for each part type. In Table 4.1, the letters N, E, S, W, F and B are used to refer to the values of the input signals at the $NORTH$, $EAST$, $SOUTH$, $WEST$, $FRONT$ and $BACK$ faces of a part. In these diagrams, $NORTH$ is up the page, $SOUTH$ is down the page, $EAST$ is to the right of the page, $WEST$ is to the left of the page, $FRONT$ is out of the page and $BACK$ is into the page. The Boolean $\neg$ (negation) and $\vee$ (OR) operators are used in Table 4.1.

The *Wire*, *Nor* and *Rotate* parts have rotational symmetry about their primary axis and can therefore be in any one of 6 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 4.2 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.

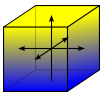
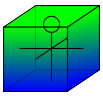
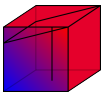
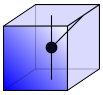
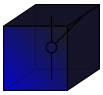
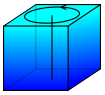
1 Wire	2 Nor	3 Slide
		
$N, E, W, F, B := S$	$N := \neg(E \vee S \vee W \vee F \vee B)$	If S then move the part that lies $NORTH$ one unit $EAST$
4 Fuse	**5 UnFuse**	**6 Rotate**
		
If S then connect the parts that lie $NORTH$ and $NORTH\text{-}EAST$	If S then disconnect the parts that lie $NORTH$ and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

Table 4.1. Part types in CBlocks3D.

Note that for the representations given in Table 4.2 $NORTH$ is up the page and $FRONT$ is out of the page. In later diagrams in this thesis 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.

In accordance with questions 1 and 2 on page 63 the main criterion that influenced the choice of this set of part types was that it should contain as little redundancy as possible yet still span all of the features that the environment supports. In section 3.3 we speculated that only 5 types of part might be required. Here we have chosen to use two distinct part types for signal propagation and logical operations rather than combine these functions into a single part type. It was found that by doing this circuits were considerably simpler and signal routing was made easier.

This set of part types is considerably simpler than that used in the CBlocks environment and comparable to the complexity of the sets used by von Neumann and Codd, but with the addition of parts required for manipulating the kinematic features of the CBlocks3D environment. It is therefore likely that this is among the simplest possible part sets in this particular environment that spans all of the features that the environment supports.

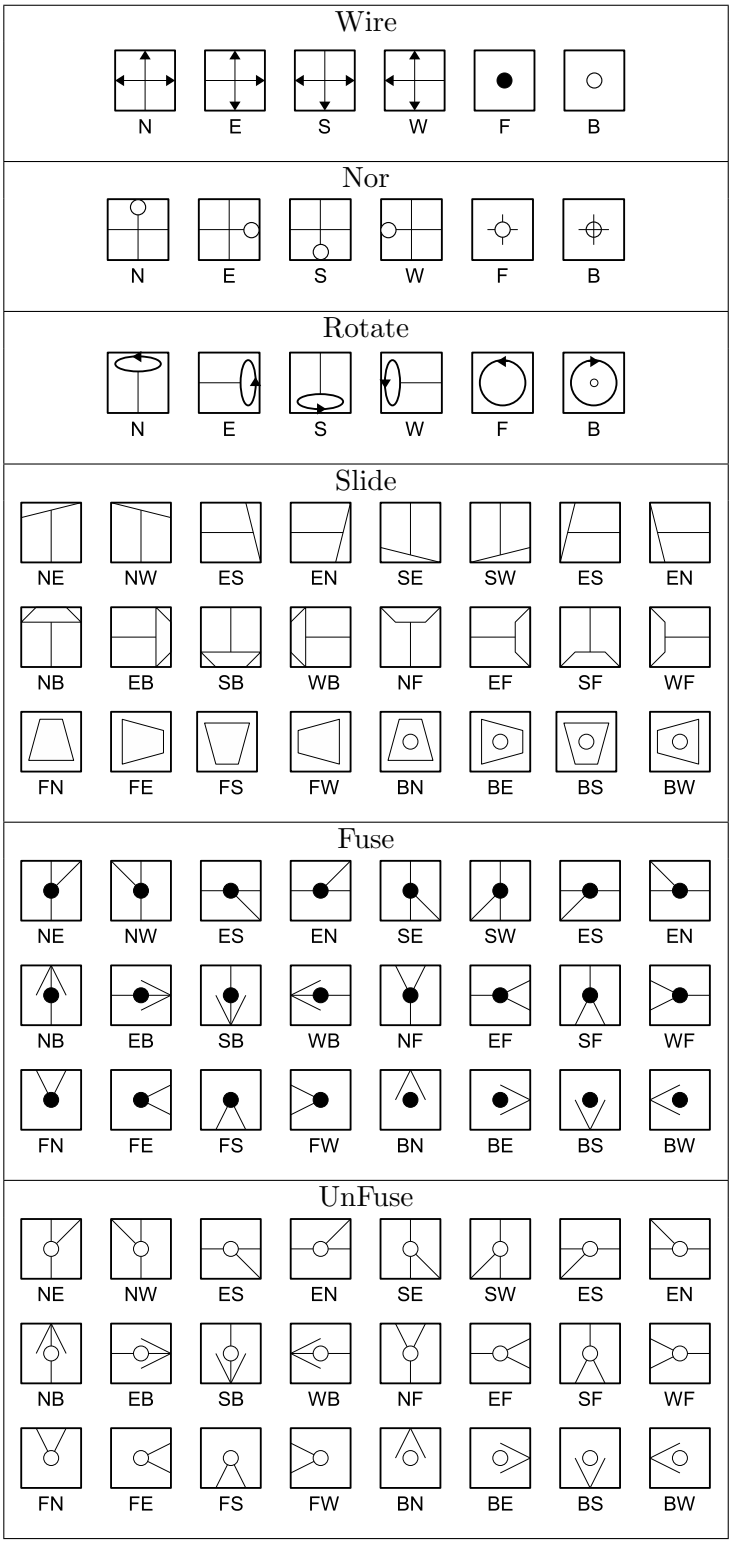


Table 4.2. Graphical representations of parts.

4.2.1 Evolution of a universe

Before defining the algorithm for the evolution of the CBlocks3D universe, the following definitions are needed.

If $p = ((x, y, z), P, S, T, O, C)$ and $V = (x', y', z') \in D$ then $p + V = ((x + x', y + y', z + z'), P, S, T, O, C)$.

We define the neighbour relation $\parallel$ for parts p_1, p_2

$$p_1 \parallel p_2 \text{ if and only if } p_1.location - p_2.location \in D$$

And the indirect-neighbour relation $\parallel_I$ for parts p_1, p_2

$$p_1 \parallel_I p_2 \text{ if and only if } p_1 \parallel p_2$$
$$\text{or there exists } p_3 \text{ such that } p_1 \parallel p_3 \text{ and } p_3 \parallel_I p_2$$

It is also useful to have a neighbour predicate which is true if and only if p_2 is a neighbour of p_1 in direction A

$$\text{neighbour}(p_1, p_2, A) \text{ if and only if } p_2.position - p_1.position = A$$

We define the directly-connected relation $\bowtie_D$ for parts p_1, p_2

$p_1 \bowtie_D p_2$ if and only if there exists $A \in D$ such that
$$\text{neighbour}(p_1, p_2, A) \text{ and } p1.connect[A] = p2.connect[\text{opposite}(A)] = True$$

And the indirectly-connected relation $\bowtie$ for parts p_1, p_2

$$p_1 \bowtie p_2 \text{ if and only if } p_1 \bowtie_D p_2$$
$$\text{or there exists } p_3 \text{ such that } p_1 \bowtie_D p_3 \text{ and } p_3 \bowtie p_2$$

The *dilate* function is used to calculate which parts from set S should move when a part p is moved in direction d by a *slide* part. The parts that should move are parts that are indirectly connected to p and also those parts that move as a result of being pushed along by other parts.

$$\text{dilate}(p, d, S) = \{q \in S \mid q = p \text{ or}$$
$$\text{there exists } r \in \text{dilate}(p, d, S) \text{ such that neighbour}(r, q, d) \text{ or } q \bowtie r\}$$

A universe evolves through an infinite sequence of states $S_0, S_1, S_2, \ldots$

S_0 completely determines subsequent states. S_{n+1} can be determined from S_n using the the algorithm given below:

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.

A formal description of this algorithm follows. This algorithm adds parts to the temporary sets I_n, J_n, A_n and $M_{d,n}$ where $d \in D$, in the course of computing S_{n+1} from S_n.

Step 1

For each part p in S_n

Set $p' = p$

If $p.type = Wire$

If there exists a part q such that

neighbour$(q, p, p.primary)$ and $q.output[p.primary] = True$

$p'.output = (True, True, True, True, True, True)$

$p'.output$[opposite$(p.primary)$] $= False$

else

$p'.output = (False, False, False, False, False, False)$

If $p.type = Nor$

If there exists a part q such that neighbour(p, q, d)

and $d \neq p.primary$ and $q.output$[opposite(d)] $= True$

$p'.output = (False, False, False, False, False, False)$

else

$p'.output = (False, False, False, False, False, False)$

$p'.output[p.primary] = True$

If ($p.type = Slide$ or $p.type = Rotate$

or $p.type = Fuse$ or $p.type = UnFuse$) and there exists a part q

such that neighbour$(q, p, p.primary)$ and $q.output[p.primary] = True$

Add p' to A_n

Add p' to I_n

Step 2

For each part p in I_n

Set $p' = p$

For every part q in A_n such that neighbour$(q, p, q.primary)$

If $q.type = Rotate$

Rotate p' 90 degrees in a right-hand direction about $q.primary$

else if $q.type = Fuse$ and there exists a part r in I_n

such that neighbour$(p, r, q.secondary)$

$p'.connect[q.secondary] = True$

else if $q.type = Unfuse$ and there exists a part r in I_n

such that neighbour$(p, r, q.secondary)$

$p'.connect[q.secondary] = False$

For every pair of parts q, r in A_n such that

neighbour$(q, r, q.primary)$ and neighbour$(r, p, q.secondary)$

If $q.type = Fuse$

$p'.connect[\text{opposite}(q.secondary)] = True$

else if $q.type = Unfuse$

$p'.connect[\text{opposite}(q.secondary)] = False$

Add p' to J_n

Step 3

For each part p in J_n

If there exists a part q in A_n such that neighbour$(q, p, q.primary)$

and $q.type = Slide$

Add $dilate(p, b, J_n)$ to $M_{b,n}$, where b = q.secondary

Add all parts in J_n but not in any $M_{d,n}$ to S_{n+1}

Step 4

For each part p in $M_{d,n}$

Add $p + d$ to S_{n+1}

In Step 2 it is illegal for a part to be operated on by more than one *rotate* part at any time, and it is illegal for both a fuse and unfuse part to operate on the same connection at the same time. In Step 3 it is illegal for a part to be in both $M_{d,n}$ and $M_{c,n}$ where $c \neq d$.

It is also illegal for any two distinct parts in S_n to have the same location.

These 'illegal' conditions are specified to avoid having to deal with conflict situations that would arise, for example, if an attempt were made to slide a part in two different directions at the same time or if an attempt were made to move two parts into the same location. Of course, rules could be written to resolve such conflict situations (as Arbib does in [4]), but since such situations do not arise in any of the mechanisms described in this thesis, and since a description of these conflict resolution rules would be lengthy, we simply define these situations as illegal and rule that no mechanism should violate these conditions.

4.3 Simple mechanisms

In this section several simple mechanisms are described which are used as the building blocks of more complex structures in Chapter 5.

4.3.1 Signals and logical values

There are several different ways in which signals, signal paths and logic gates can be used for representing and processing logical values. It is necessary here to distinguish between the *True* and *False* values that the outputs of individual *wire* and *nor* parts can take, and the logical value being represented by a collection of parts. We will use the terms 'logic 0' and 'logic 1' to refer to the value being represented by a collection of parts.

One way of representing logical values using a collection of parts is to stipulate that an entire signal path connecting one logic element to the next represents a single logical value. If all of the *wire* parts in the signal path are outputting a *True* signal then the path represents a logic 1 value. If none of the wires in the signal path are outputting a *True* signal then the path represents a logic 0 value. Other states of the path represent a transition from one logic value to another. This representation will be referred to as the *static signal* representation.

Alternatively we could stipulate that the presence of a *True* signal at a particular location at a particular time represents a logic 1 value and its absence represents a logic 0 value. The advantage of this approach is that a sequence of successive values can be sent serially into one end of a signal path and processed or decoded in serial fashion. The disadvantage of this approach is that a timing signal must be present somewhere within a circuit to indicate when the output of a particular part should be sampled. The timing signal can be sent along a separate pathway, or it could be a signal that precedes the

signal that represents the logical value on the same signal path. This representation will be referred to as the *pulse signal* representation.

It is also useful to employ a third representation that is similar to the *pulse signal* representation, but in which a logic 1 value is represented by a sequence of 4 consecutive *True* values on a signal path. The reason for this is that some circuit elements require the presence of a signal on an input for 4 time units in order for the signal to propagate throughout the element before the input changes. This representation will be referred to as the *4-pulse signal* representation.

4.3.2 Logic gates

Regardless of which of these representations is used in a circuit, the structure of combinatorial logic gates is the same. Figure 4.1 shows how the *Nor* part can be used as the basis for other logic gates.

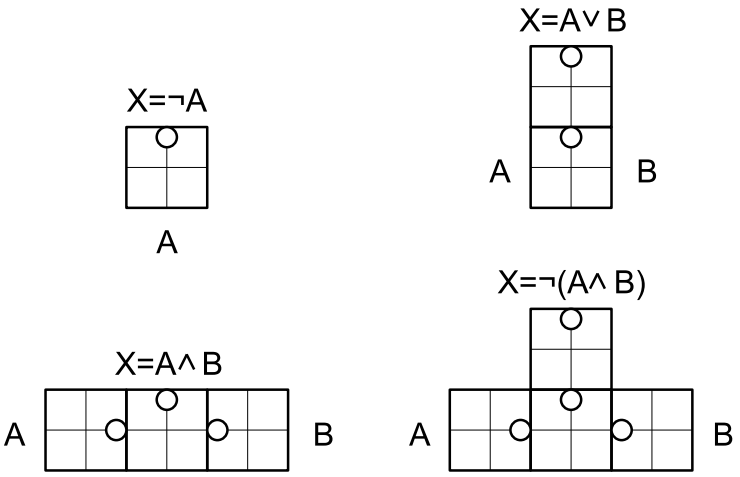


Figure 4.1. NOT, OR, AND and NAND gates made from NOR gates.

If a *pulse signal* or *4-pulse signal* representation is used then circuits must be arranged so that values arriving at all inputs to a gate are synchronised with each other.

For some gates it is possible to use different representations for different inputs. For example, if input A of an AND gate uses the *static signal* representation and input B uses the *pulse signal* representation then the output from the AND gate will be in the *pulse signal* representation.

4.3.3 Edge detection

Figure 4.2 shows a circuit that will output a *4-pulse signal* logic 1 value at X when the value on *static signal* path A changes from logic 0 to logic 1.

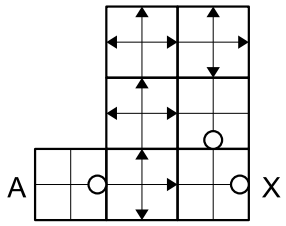


Figure 4.2. Circuit for detecting the rising edge of a signal.

4.3.4 Signal loops

A loop of wire parts such as that shown in Figure 4.3 can be used to store a sequence of signals. The signals will propagate around the loop endlessly.

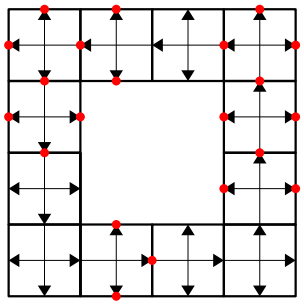


Figure 4.3. A sequence of signals stored in a loop of wire parts.

The addition of two *nor* parts to this signal loop (Figure 4.4) 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. Figure 4.5 shows how two gated signal loops can be held against one another so that signals from one can be copied into the other.

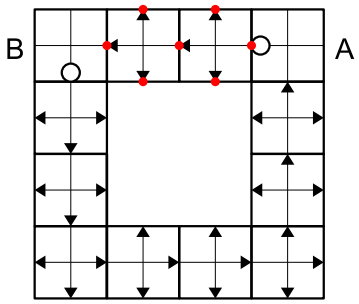


Figure 4.4. A gated signal loop incorporating two *Nor* parts.

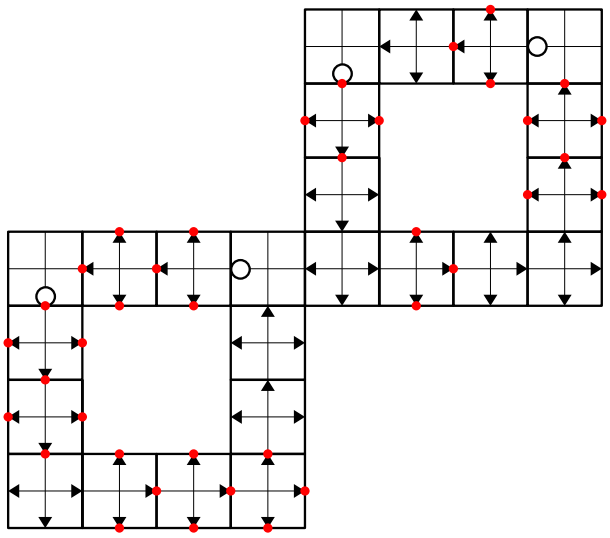


Figure 4.5. A sequence of signals being copied from one loop to another.

4.3.5 Flip flops

A Set-Reset flip flop is a circuit element which, when it receives a logic 1 value on its *Set* input, will enter a state in which it outputs a logic 1 value, and when it receives a logic 1 value on its *Reset* input will enter a state in which it outputs a logic 0 value. If both *Set* and *Reset* inputs receive a logic 1 value at the same time then the behaviour of the element is not defined, so circuits must be designed so that this does not occur.

Figures 4.6 and 4.7 show two different flip flop designs, each of which uses a different representation for its inputs and has different properties that determine its suitability in a given context.

The flip flop shown in Figure 4.6 can use the *static signal* or *4-pulse signal* representations for its inputs. In order for a signal to propagate through the flip-flop and alter its

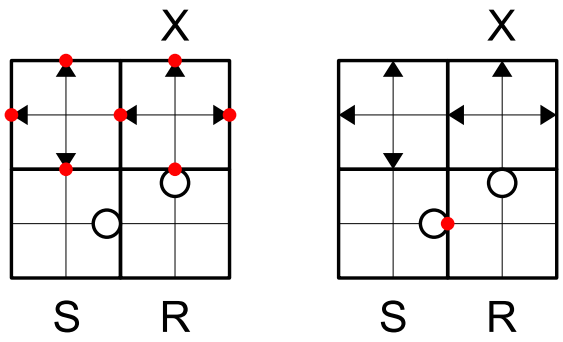


Figure 4.6. A Set-Reset flip-flop made from *Wire* and *Nor* parts.

state, a signal must be present on an input for a minimum of 4 time units. The output of the flip-flop uses the *static signal* representation.

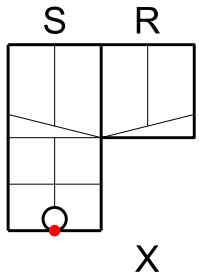


Figure 4.7. A Set-Reset flip-flop that makes use of *Slide* parts.

The flip flop shown in Figure 4.7 can use any of the three signal representations for its inputs. This flip flop is able to respond to a pulse 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. If it were then the whole mechanism would move $EAST$ or $WEST$, rather than just the *nor* part. If this flip-flop is used in a machine that moves around in its environment then the *nor* part could be left behind because the *nor* part is not connected to the rest of the mechanism. To solve this problem, the *nor* part could be enclosed by a cage made from other parts to prevent it from being lost, or an additional mechanism using *fuse* and *unfuse* parts could be employed to make sure that the *nor* part is connected to the rest of the mechanism whenever neither input is asserted.

4.3.6 1:4 pulse converter

In a circuit that uses a mixture of signal representations it may be necessary to use a pulse converter to convert a pulse 1 time unit long into a pulse 4 time units long. The circuit element shown in Figure 4.8 does this: a *pulse signal* logic 1 at input A will be converted to a *4-pulse signal* logic 1 at output B. The mechanism makes use of a *slide* part to respond to a pulse 1 time unit long. The mobile *nor* part C in Figure 4.8 is normally connected to the rest of the mechanism. It is only disconnected when a pulse arrives, and then reconnected again when the mechanism has finished its operation.

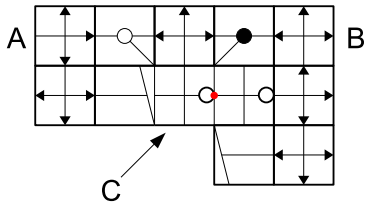


Figure 4.8. 1:4 Pulse Converter for converting between *pulse signal* and *4-pulse signal* representations.

4.3.7 Transporting parts

Figure 4.9 shows how *slide* parts can be used to make a path along which other parts can travel. Each of the *slide* parts in Figure 4.9 has a *nor* part providing it with a *True* signal. The dashed square stands for any part.

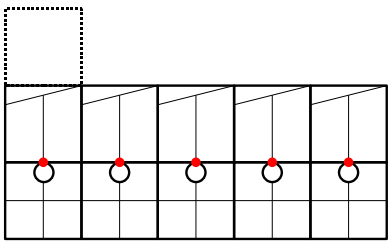


Figure 4.9. A path along which parts can travel.

If any of the *nor* parts in Figure 4.9 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.

4.3.8 Encoders and decoders

In a large circuit it may be necessary to transmit a signal from one region of the circuit to a distant region to activate an element in the distant region. If there is a need for several such signals then the number of signal paths leading from one part of the circuit to the other can be reduced by multiplexing the signals onto a single path. There are a number of different possible ways of multiplexing signals. The method described here uses an encoding scheme in which each signal is assigned a unique encoding which is transmitted serially along the communication path and decoded at the other end. This method can be used when it can be guaranteed that the interval between any pair of signals being asserted is longer than the time taken to encode the signal onto the communication path.

The left-hand circuit in Figure 4.10 is an encoder that will output at X the sequence 101010101 when a True signal is input at A. Each of the parts labelled B_0, B_1 and B_2 may be omitted to alter the sequence that will be output, so that any of the sequences $10B_00B_10B_201$ can be generated.

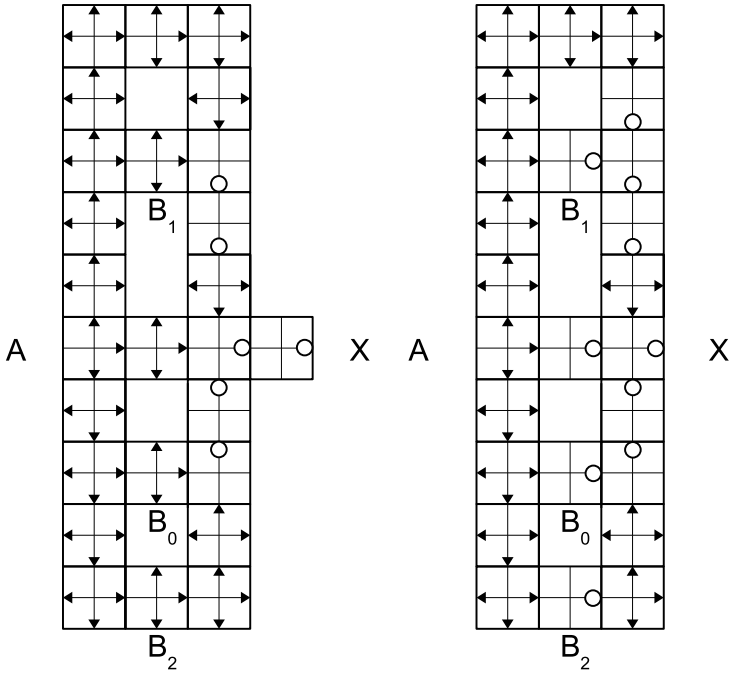


Figure 4.10. An encoder and decoder for generating and detecting serial signal sequences.

The right-hand circuit in Figure 4.10 shows the corresponding decoder circuit. By choosing either a *wire* or *nor* part for each of the parts B_0, B_1 and B_2, any of the sequences $10B_00B_10B_201$ can be decoded.

4.3.9 Counters and registers

Figure 4.11 shows how 11 parts can be added to the flip-flop of Figure 4.6 to turn it into a toggle flip-flop which can be used as the basis for an asynchronous counter. A *4-pulse* logic 1 at the T input will cause the flip-flop to change state.

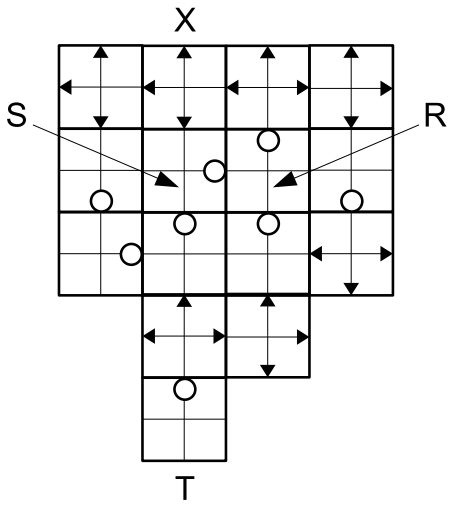


Figure 4.11. A toggle flip-flop.

The faces that were used as *Set* and *Reset* inputs in Figure 4.6 are no longer available in Figure 4.11. 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 Figure 4.11).

A convention that will be used for diagrams of three-dimensional structures later on in this thesis is introduced in Figure 4.12. All previous diagrams have been of planar circuits. Figure 4.12 shows a three dimensional circuit. The circuit can be thought of as being a 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 4.12 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). V uses the *static signal* representation and L uses the *4-pulse signal* representation. A *4-pulse* logic 1 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 toggle input of the next, we can make an N-bit asynchronous

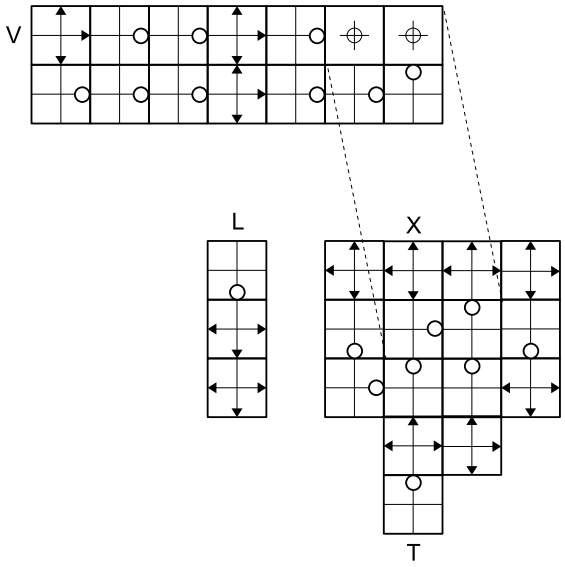


Figure 4.12. A toggle flip-flop with Value V and Load L inputs.

counter that will increment in response to a *4-pulse signal* clock on the $Toggle$ input of the first flip-flop. One disadvantage of this type of counter is that it is slow to count up. For example, when an N-bit counter rolls over from $2^N - 1$ to 0, a signal has to propagate from one end of the counter to the other through every flip-flop and edge detection circuit. A second disadvantage is that if we wish to make a loadable counter in which a value can be loaded into the counter in a similar manner to the way in which a value can be loaded into the flip-flop of Figure 4.12, then the negative edge detectors must somehow be inhibited during the loading of a value into the counter. If this were not done then changes in the output from any flip-flop from logic 1 to logic 0 could cause unwanted toggling of subsequent flip-flops.

A well-known way of improving this situation is to incorporate logic into the counter that precalculates which bits of the counter will need to toggle in response to the next clock signal. A picture of such a counter in CBlocks3D is shown in Figure 4.13.

4.3.10 Memory

There are a number of different ways in which information storage structures may be implemented using the part types available. One way to do this would be to use logic gates and registers made from flip-flops to make memory circuits similar to those found in electronic computers. If we define the *storage density* of a memory as the number of

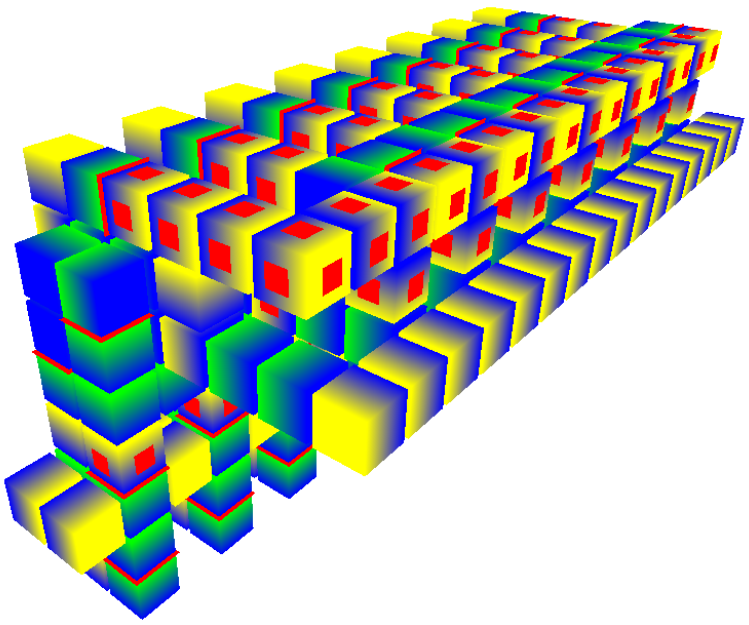


Figure 4.13. An 8-bit loadable synchronous counter.

bits of information that can be stored per unit volume, then such a memory would have a low density. The 3-part flip flop shown in Figure 4.7 can store a single bit, so even before taking into account decoding and multiplexing logic, the storage density is only 0.33 bits per unit volume.

An alternative memory architecture can be based around a signal loop. We have already seen in section 4.3.4 that a pattern of signals will propagate endlessly around such a loop, which has some similarity to the delay line stores that were used in early computer memories. 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 any conceivable 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.

The volume may be empty, or it may contain one of the three axially symmetric part types (two of which, the *wire* and *nor* part types may be in one of two states) in any one of 6 possible orientations, or it may contain one of the three axially asymmetric parts in any one of 24 possible orientations. The total number of possible states for a unit volume is therefore $1 + 30 + 72 = 103$, requiring $\log_2 91 = 6.7$ bits to describe. A part may be connected to neighbouring parts. There are at most 3 connections per part, each

requiring one bit to describe. Therefore the largest possible memory density is 9.7 bits per unit volume. This analysis gives a slight overestimate for the upper limit because we ignored the fact that no connection can be made to a volume containing no part, and that in practice maintaining a *wire* or *nor* cell in an excited state independently of the state of its neighbours would not be possible.

This upper limit is not a practically realistic figure. Although it is possible to determine whether a particular part is connected to its neighbours by sliding it and seeing whether it separates from its neighbours, it is not possible in general to tell which of its neighbours it is connected to. Also, it is impractical to construct a storage system that contains *nor* parts because if the active face of a *nor* part lies against the input of a *rotate*, *fuse*, *unfuse* or *slide* part, then unwanted activity would occur. Additionally, if the memory system is to form part of a system that is mobile, it is not wise to permit cells within it to be empty, since particular patterns of information could result in a memory containing structures surrounded by empty cells. These structures would end up out of place when the rest of the memory moved.

Therefore, let us consider a memory in which information is encoded using the type and orientation of parts in a solid volume of parts where we restrict ourselves to *wire*, *slide*, *fuse*, *unfuse* and *rotate* parts. Here there are two axially symmetric parts and three non-symmetric parts, so the total number of possible states for a cell is $12 + 72 = 84$, giving a memory density of 6.4 bits per unit volume.

Reading information from a particular address in a memory constructed in this way would be a matter of extracting the parts corresponding to the memory address to be read, then examining the parts to determine their type and orientation. A mechanism for determining the type of a part is also needed by an SRPC and is described in section 5.4. This mechanism could be modified to support determination of the orientation of a part based on counting the number of passes through the mechanism of section 5.4 that are required to identify the part.

Writing information into a memory of this type is a construction operation identical to the operation performed by the orientation and constructing subsystems described in Sections 5.6 and 5.7.

However, despite the high density of this type of memory and the fact that some of the mechanisms required for reading and writing this type of memory are required elsewhere in a constructing automaton, and could perhaps be shared to avoid duplication of identical mechanisms, 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.

4.4 Methodology and implementation

4.4.1 Simulating the CBlocks3D environment

The algorithm given in section 4.2.1 is written with the aim of giving as clear as possible a description of the behaviour of the CBlocks3D environment. Although this algorithm could be implemented exactly as it is written, the resulting implementation would not be efficient because it does not take account of any repetition or redundancy in the system being simulated.

A more efficient implementation that draws upon some of the techniques used by the *Hashlife* algorithm published in [25] is described here. This implementation automatically detects when structures within the system being simulated are repeated in space and avoids simulating them twice. It also detects when a structure is in a state that has already been simulated and avoids the need to simulate it again by storing the results of previous simulation steps.

An octree data structure is used to represent the state of a CBlocks3D universe. In this representation a cubic volume of space large enough to contain the system being represented is subdivided into eight smaller cubes. Each smaller cube is subdivided in a similar way down to the level of individual parts. This is implemented as a tree data structure where each node of the tree corresponds to a cubic volume and contains an array of eight pointers, each corresponding to a smaller cube. At any level within this tree any two cubic volumes may be identical and can be represented by the same node. In this way regions of empty space and repeated sub-structures can be represented efficiently. To illustrate this, Figure 4.14 shows the two-dimensional version of this representation (known as a quadtree). In this figure different coloured lines are used to represent pointers from a node in one level to a node in a lower level. Red lines represent pointers to the top left sub-square of a node, green lines represent pointers to the top right, blue to the bottom left and black the bottom right.

The *Hashlife* algorithm is so named because a large hash table is used to keep a collection of pointers to nodes which have already been used. When a new node is built as the result of a simulation step it is looked up in the hash table to see whether it has already been used elsewhere in the octree and also whether it has been used some time in the past.

The major difference between the CBlocks3D environment and cellular automata is that in CBlocks3D connected structures can move as one piece in a single time step. The *Hashlife* algorithm requires that any spatial volume is affected only by nearby volumes,

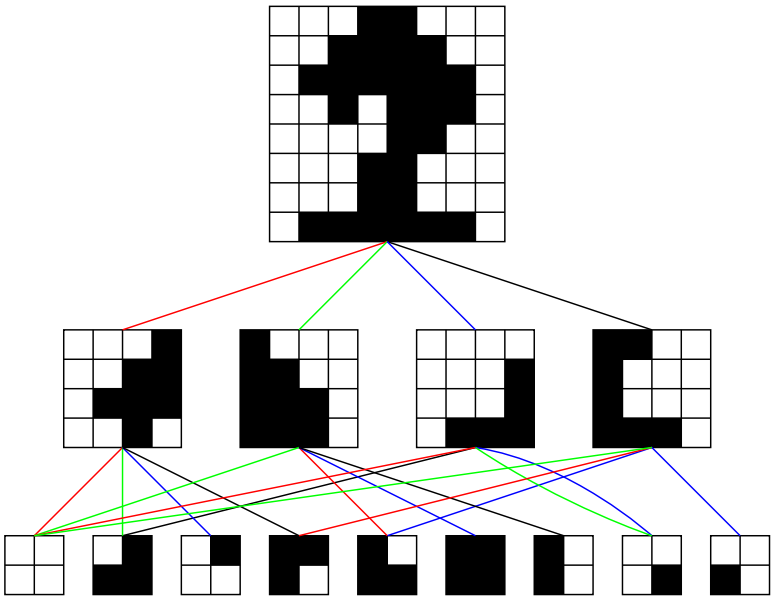


Figure 4.14. A quadtree data structure.

so is well suited to simulating cellular automata but not the CBlocks3D environment. To overcome this difficulty, simulation of CBlocks3D is split into two interleaved phases called the stationary phase and the movement phase, each of which can be independently simulated by the *Hashlife* algorithm. During the stationary phase the activity of all parts except the *slide* part are simulated. The stationary phase runs until the point when a *True* signal is applied to any *slide* part with a part against its active face. At this point movement is about to occur and the simulation switches to the movement phase. During the movement phase a cellular automaton model is used to propagate the effect of the movement to all affected parts. At the end of the movement phase the affected parts move and simulation switches back to the stationary phase.

The same octree representation is used regardless of which phase is being simulated. The only difference between the two phases is the set of rules used to update the state of parts.

To carry out a single simulation step the parts within a cubic volume of dimension 2^n are used to compute a future state of a volume of dimension 2^{n-1} centred on the original volume. Doing this throughout the octree would leave gaps between volumes in the computed result. To overcome this several overlapping octrees are constructed at the time that a calculation at a particular level in the octree needs to be carried out so that the results from each can be combined in such a way that there are no gaps. Whenever

a calculation is carried out on an octree node, the result of the calculation is stored in a special result pointer within the node so that if the same calculation is needed in future it does not need to be carried out again, but can be retrieved from this pointer.

In this implementation of the *Hashlife* algorithm the state of the environment 8 time units ahead is computed, unless it is detected that a transition to a different phase is needed at some time within those 8 time units, in which case simulation is backtracked and then stepped forward 1 unit at a time.

A program listing is given in appendix A and included on the CD attached to this thesis.

This implementation allows us to simulate the 225 million iterations that are required for the SRPC described in Chapter 5 to construct a replica machine in an initial environment containing enough parts to construct a replica. The simulation was run in a little over 10 days on an Intel Core 2 Duo processor running at 2.6 GHz with 4Gb of RAM.

4.4.2 Describing structures

To describe complex three dimensional structures made from connected substructures a suite of C++ classes is used. In the same way that the part-level diagrams used in this chapter and Chapter 5 are drawn layer by layer, these classes allow small structures to be described layer by layer. For example the following listing shows how a structure can be built up layer by layer from $SOUTH$ to $NORTH$. Outputs of parts within these layers can be set and then terminals added for connecting this structure to other structures. For historical reasons, the three letter abbreviation *del* is used to specify a *wire* part.

```
void SyncStageN(CBasicObject &bo)
{
  bo.Layer();
  bo.String("{nul {n e}} {nul {n e}}");
  bo.String("{nul {n e}} {nul {n e}}");
  bo.String("{nor {b e}} {nul {n e}}");
  bo.String("{del {w n}} {del {w n}}");
  bo.Layer();
  bo.String("{nul {n e}} {nul {n e}}");
  bo.String("{nul {n e}} {nul {n e}}");
  bo.String("{del {s e}} {nul {n e}}");
  bo.String("{nul {n e}} {nul {n e}}");
  bo.Layer();
  bo.String("{del {f e}} {nul {n e}}");
```

```
  bo.String("{nor {e f}} {nor {f e}}");
  bo.String("{del {e f}} {del {f n}}");
  bo.String("{nul {n e}} {nul {n e}}");

  bo.SetOutputs(0,2,0,true);
  bo.SetOutputs(0,2,2,true);
  bo.SetOutputs(1,2,1,true);
  bo.SetOutputs(1,2,2,true);
  bo.SetOutputs(0,1,2,true);

  bo.SetOutputs(0,0,3,true);
  bo.SetOutputs(1,0,3,true);

  bo.Terminal("out",0,0,2,false,back);
  bo.Terminal("count-in",1,0,3,true,east);
  bo.Terminal("count-out",0,0,3,false,west);
}
```

The next listing shows how more complex structures can then be put together from simpler structures.

```
void CounterN(CCompoundObject &co, unsigned n)
{

  CBasicObject *ff;
  CBasicObject *ss;

  ff = new CBasicObject [n];
  ss = new CBasicObject [n];

  {for(unsigned i = 0; i<n; i++)
  {
    SmallToggleV2(ff[i]);
    if (i)
      SyncStageN(ss[i]);
    else
      SyncStage0(ss[0]);

    ff[i].Rotate(back);
    ff[i].Rotate(west);
```

```
  }}

  co.AddObject(&ff[0],"ff0",0,0,0);
  co.AddObject(&ss[0],"ss0","out","ff0.t");

  {for(unsigned i = 1; i<n; i++)
  {
    char tmp[16],tmp2[16];

    sprintf(tmp,"ss%d",i);
    sprintf(tmp2,"ss%d.count-in",i-1);

    co.AddObject(&ss[i],tmp,"count-out",tmp2);

    sprintf(tmp,"ff%d",i);
    sprintf(tmp2,"ss%d.out",i);

    co.AddObject(&ff[i],tmp,"t",tmp2);
  }}
}
```

After a structure has been specified it can be written to a file. The file format used contains not only a description of the parts within a structure, but also the names that were given to substructures when the structure was specified. A separate file containing the names of all terminals within the structure can also be generated so that test stimuli can be applied to some terminals and outputs from others monitored.

4.4.3 Visualisation and debugging

Visualisation of three dimensional structures is invaluable for working out how to fit them together and for helping to uncover design errors in large complex structures.

A utility was written using the OpenGL graphics library to render structures and to help with debugging. The utility allows users to navigate around (or even inside) 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.

The utility allows the user to specify a project file with a `.prj` file extension which contains a list of other files to load. The following list shows what each type of file is used for:

`.grd` Specifies the structure to be loaded.

`.trm` Lists terminals within the structure. These correspond to terminals created using the Terminal function when specifying a structure.

`.sti` Lists stimulus inputs and the terminals that they should be applied to.

`.uvw` Contains locations and orientations of views of the structure.

`.mon` Lists terminals to monitor as simulation progresses. Allows a 'trigger' terminal to be specified, so that values from a collection of terminals will only be output when the value of the trigger terminal is *True*.

The CD attached to this thesis contains examples of each type of file, along with a description of the file formats used.

4.5 Summary

This chapter has described a 3D kinematic environment that supports 6 simple types of part and has shown how parts can be put together to make simple circuits and mechanisms. The next chapter shows how these mechanisms and others can be used to build subsystems that can then be put together to make a self-replicating programmable constructing machine.

Chapter 5

A Self-Replicating Programmable Constructor in the CBlocks3D environment

5.1 Design considerations

Figure 5.1 shows a black-box diagram of a programmable constructing machine in the CBlocks3D environment. The machine has an input orifice that receives parts from the machine's environment. The machine outputs a construction which is produced from an internal description or program without any external signals being fed into the machine.

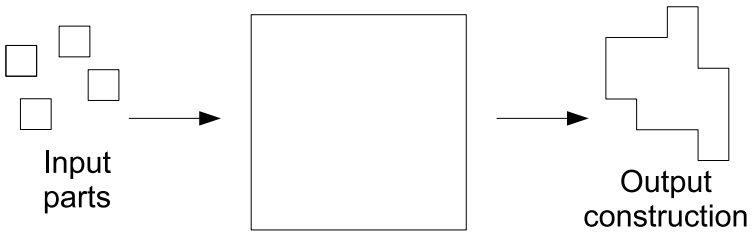


Figure 5.1. Black box diagram of a programmable constructing machine.

We shall regard the operation of a machine constructing another object as a task that includes the task of obtaining the parts needed to construct that object from the machine's environment.

By considering what happens to an individual part passing through the black box from the input orifice to the object being constructed, we can deduce what some of the essential

subsystems are that the constructing machine must contain, and also what the possible variations are in how the subsystems can be arranged. We shall also find that there are some subsystems that are not essential but which if included will decrease the time taken for the machine to carry out a construction operation.

The first simplifying assumption that we will make (and which is implied in Figure 5.1) is to assume that the machine's construction operations are serial. This means that there is a single location at which construction occurs one part at a time. We shall also assume that the machine's processing of input parts from its environment occurs serially with one part being processed before another part is obtained from the environment.

5.1.1 Managing parts

The construction output of the machine outputs a particular series of parts with each part in a particular orientation. At its input the machine receives parts in an unknown order with each part in an unknown orientation. There are several different ways of ensuring that the correct part in the correct orientation is available when the machine needs to output it.

One method is to discard all parts arriving at the input until the correct part in the correct orientation arrives. This method assumes that both the type and orientation of input parts are distributed in a way that guarantees that the correct part in the correct orientation will eventually arrive.

Another method is to discard all parts until a part of the correct type in any orientation arrives and then alter the orientation of that part if necessary. This method makes no assumption about the distribution of the orientation of input parts.

A third method is to identify and store all parts arriving at the input, only discarding parts if the storage area is full. When the construction output requires a part it requests the part from the store. Parts could be stored in any orientation and then reorientated to the desired orientation when required. Alternatively all parts could be stored in the same orientation and then reorientated to the desired orientation when required.

An approximate analysis is carried out below to show how the first and the third methods differ in the time required to collect the parts needed for a construction. This analysis does not take into account the influence that the chosen method has on the size of the system.

Let us assume that a machine M is constructing a machine N. Let $C_N(t, o)$ be a count of the number of times that a part of type t and orientation o occurs in machine N.

Let $P_N(t) = \sum_o C_N(t, o)$ be the number of parts of type t in machine N.

Let $F_M(t, o)$ be the probability that when M encounters a part in its environment, the part is of type t and orientation o.

Let T_{Find} be the mean length of time between M beginning to search for a part in its environment and finding any part.

For the first method described above the expected length of time that machine M spends gathering parts to construct machine N is given by

$$\sum_{t,o} \frac{T_{Find} \cdot C_N(t, o)}{F_M(t, o)} \tag{5.1}$$

For the third method described above let H_p be the capacity of the storage area for part type p. Let q denote the most frequently used part. We will assume that storage areas will only be refilled when the storage area for the most frequently used part is empty.

The analysis below also assumes that parts are distributed uniformly in both the environment and the machine N being constructed. We further assume that no wastage ever occurs due to the storage area for a part being full when the part needs to be stored. The smaller the storage area size the less valid these assumptions are.

The expected amount of free space in the storage area for part t at the time that the storage area for q is empty is given by

$$\frac{P_N(t)}{P_N(q)} \cdot H_q \tag{5.2}$$

For the sake of simplifying this analysis, we will assume that the size of the storage area for each part t is at least large enough to hold this number of parts.

The expected length of time needed to obtain k parts of type p from the environment is given by

$$\frac{k \cdot T_{Find}}{\sum_o F_M(p, o)} \tag{5.3}$$

But because no part of any type is discarded until the hopper for that part type is full, the length of time needed to fill the empty space in all storage areas is the maximum length of time needed to fill any single storage area:

$$\underset{t}{Max} \frac{P_N(t) \cdot H_q \cdot T_{Find}}{P_N(q) \cdot \sum_o F_M(t, o)} \tag{5.4}$$

The expected number of times that storage areas will need to be filled when constructing N is given by

$$\frac{P_N(q)}{H_q} \tag{5.5}$$

Multiplying this by Equation 5.4 gives the expected length of time that M spends gathering parts to construct N.

$$\underset{t}{Max} \frac{P_N(t) \cdot H_q \cdot T_{Find}}{P_N(q) \cdot \sum_o F_M(t,o)} \frac{P_N(q)}{H_q} \tag{5.6}$$

This can be simplified to

$$\underset{t}{Max} \frac{P_N(t) \cdot T_{Find}}{\sum_o F_M(t,o)} \tag{5.7}$$

To express this another way: of the six part types available there is one part type for which the ratio of the number of times that it occurs in N and the probability of its occurrence in the environment is larger than the others. The total time that M spends collecting all parts needed for N is no longer than the time spent collecting this type of part.

This confirms what seems to be intuitive: storing parts for later use is in general the most efficient way to collect parts. This method also has another great advantage. If the machine had to collect another part whenever it placed a single part into the machine being constructed, it would have to make sure that the movement required to collect the next part did not interfere with the construction being carried out. One way of doing this is to make sure that the construction is always in a state where it can be attached to the parent machine and moved along with the parent when collecting parts. For this to be possible there can be no disconnected subsystems during construction. Another method is to make the part collection mechanism moveable separately from the construction part of the machine. For example the collection mechanism could be placed onto a boom that could be extended into the environment to seek out parts.

By storing parts for later use this problem is removed. The machine can operate in two phases. In one phase it carries out construction operations. Then when one of the storage areas has run out of parts the machine can attach itself to the construction and beginning collecting more parts until all of the storage areas are full. When to make the transition between the construction phase and the collection phase is something that will be known by the programmer. The programmer knows the capacity of the storage areas and therefore knows when all parts in one storage area have been used up and can issue an instruction telling the machine to begin collecting more parts. During the

construction phase the machine is free to construct in whatever manner it likes without regard for whether it is operating on disconnected subsystems. Only when the time comes to collect more parts does the machine need to ensure that all parts of the construction are connected and will not fall apart when moved.

There are several possible methods of arranging the part storage area. To avoid having unwanted interactions between parts it is reasonable to use a separate storage area for each part type. It is also reasonable to make sure that the orientation of every part within the storage area is identical. By organising the storage areas in this way we make sure that whenever a part is requested from the storage area both the type and orientation of the part are known.

5.1.2 Controlling the machine

Sections 2.2 and 2.3.1 describe two distinct architectures for the control unit of a self-replicating programmable constructor (SRPC). We can use either Thatcher's architecture and implement the control unit as a universal computer, in which case one single control unit would be used both for directing the construction operations as well as for copying the contents of the parent memory into the child machine. Alternatively we can use von Neumann's architecture in which the control unit simply interprets a sequence of instructions. In this instance a separate mechanism is employed to copy the contents of the parent memory into the child machine.

There are two considerations that lead to a preference for the second option. The primary consideration is the ease with which the contents of a memory based on signal loops can be copied into another memory. This is done by simply holding one memory against another for the length of time that it takes for signals to propagate around the loop, as described in section 4.3.4.

The second consideration is that, although the system described in this chapter is a proof-of-principle rather than a design for a practical machine, testing individual subsystems and validating the overall design is an important part of the proof. Being able to simulate the design in its entirety and show that the machine does what it is expected to do is a convincing validation. For this reason, speed of operation must be taken into account when designing the architecture of the control unit. A universal computer can be expected to require more parts to construct than a simpler control unit and might also require a longer time to execute each instruction.

In common with the design of Nobili [46], the design given in this chapter uses a method of reducing the amount of redundancy in the instruction sequence used for directing the

machine (but not the same method used by Nobili [46]). An explanation of how this was done is given in section 5.10 where the instruction encoding scheme is discussed.

5.2 Overview

Figure 5.2 shows a top-level schematic of the SRPC. The schematic shows how the system can be divided into the subsystem that manages the collection, storage, transport and positioning of parts and the subsystem that controls the whole machine.

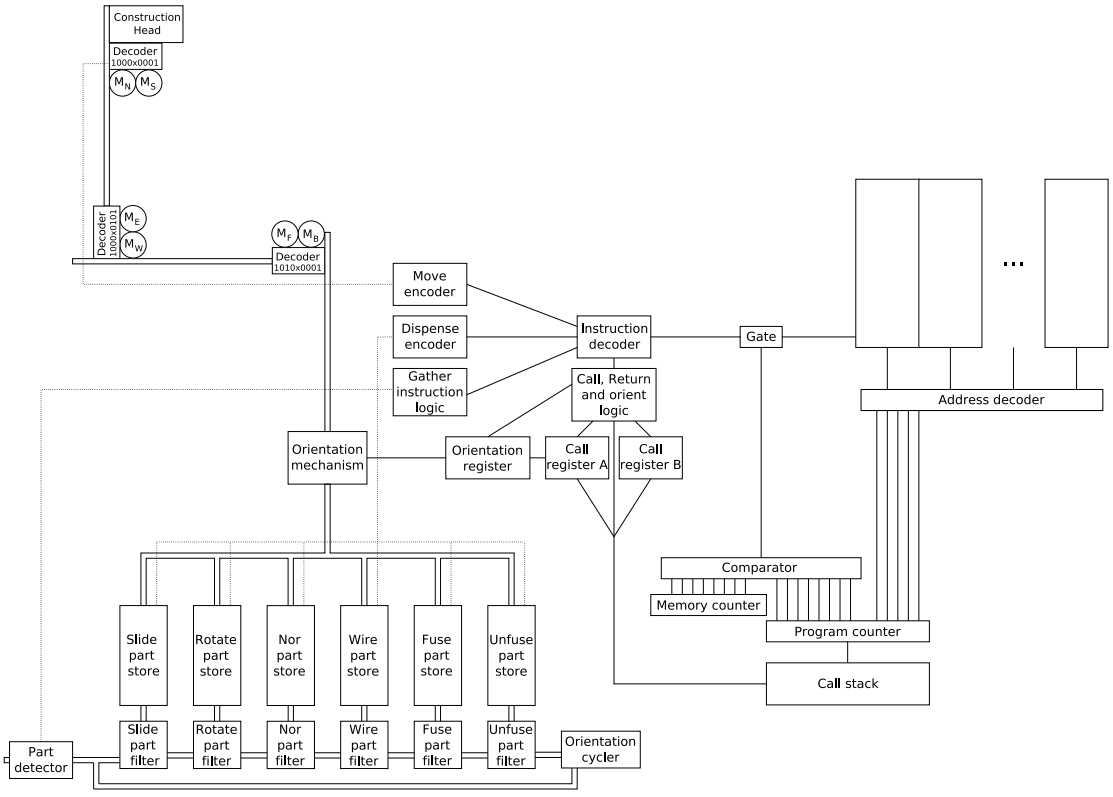


Figure 5.2. Top-level schematic of the SRPC.

Figure 5.3 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.

Each module of the system is described in a separate section in the following part of this chapter. Descriptions of some of the signal paths and logic that connect modules are omitted. The full specification of the system along with software for simulating and rendering the system are available on the CD attached to this thesis.

Three different types of diagram are used in the following sections.

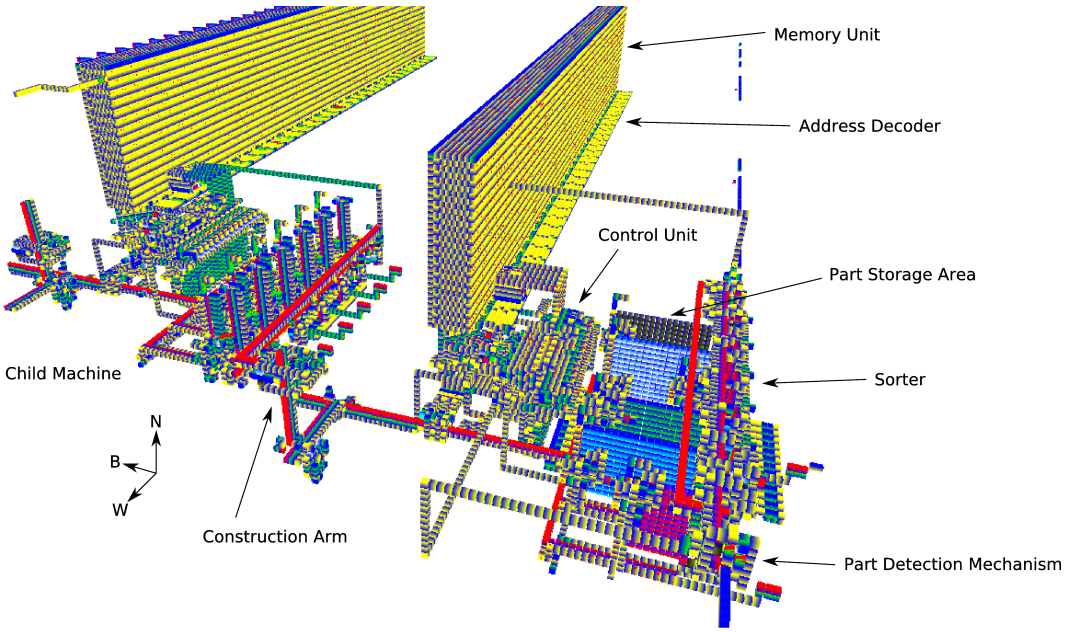


Figure 5.3. Graphical representation of the SRPC.

Schematic diagrams are somewhat like digital electronic circuit diagrams and fulfill a similar purpose. Generally these show modules 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 described in Table 4.2 on page 69. Most mechanisms have a three dimensional structure and so use the convention for displaying multi-layered structures introduced in Figure 4.12 on page 82. Unless otherwise specified all neighbouring parts in any structure are connected. The Boolean outputs from individual parts are not shown in part-level diagrams. Showing individual outputs was found to be detrimental to the utility of part level diagrams.

Graphical representations of most mechanisms are given. 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.

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

5.3 Collecting parts

When the machine 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 5.4 shows a mechanism that is able to detect when a part has entered the machine and Figure 5.5 shows a graphical representation of the mechanism. This mechanism is called the *detect* mechanism.

The *detect* mechanism has two signal inputs. One is the *part-request* input into which a logic 1 *pulse signal* is sent when the *detect* mechanism should begin looking for a part. The other input is the *enable/disable* input. This input uses the *static signal* representation. When this input is set to logic 1 it prevents the *part-request* input from having any effect and also stops any current activity of the *detect* mechanism. When the *enable/disable* input transitions from logic 1 to logic 0, it has the effect of both enabling the mechanism and also injecting a logic 1 *pulse 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 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.

5.4 Sorting Parts

A description of a mechanism for sorting parts in the CBlocks3D environment was published in reference [64]. This section elaborates on that description.

Figure 5.6 is a schematic diagram of the *sorter* mechanism which also shows how it is connected to the *detect* mechanism. Figure 5.7 shows a graphical representation of both

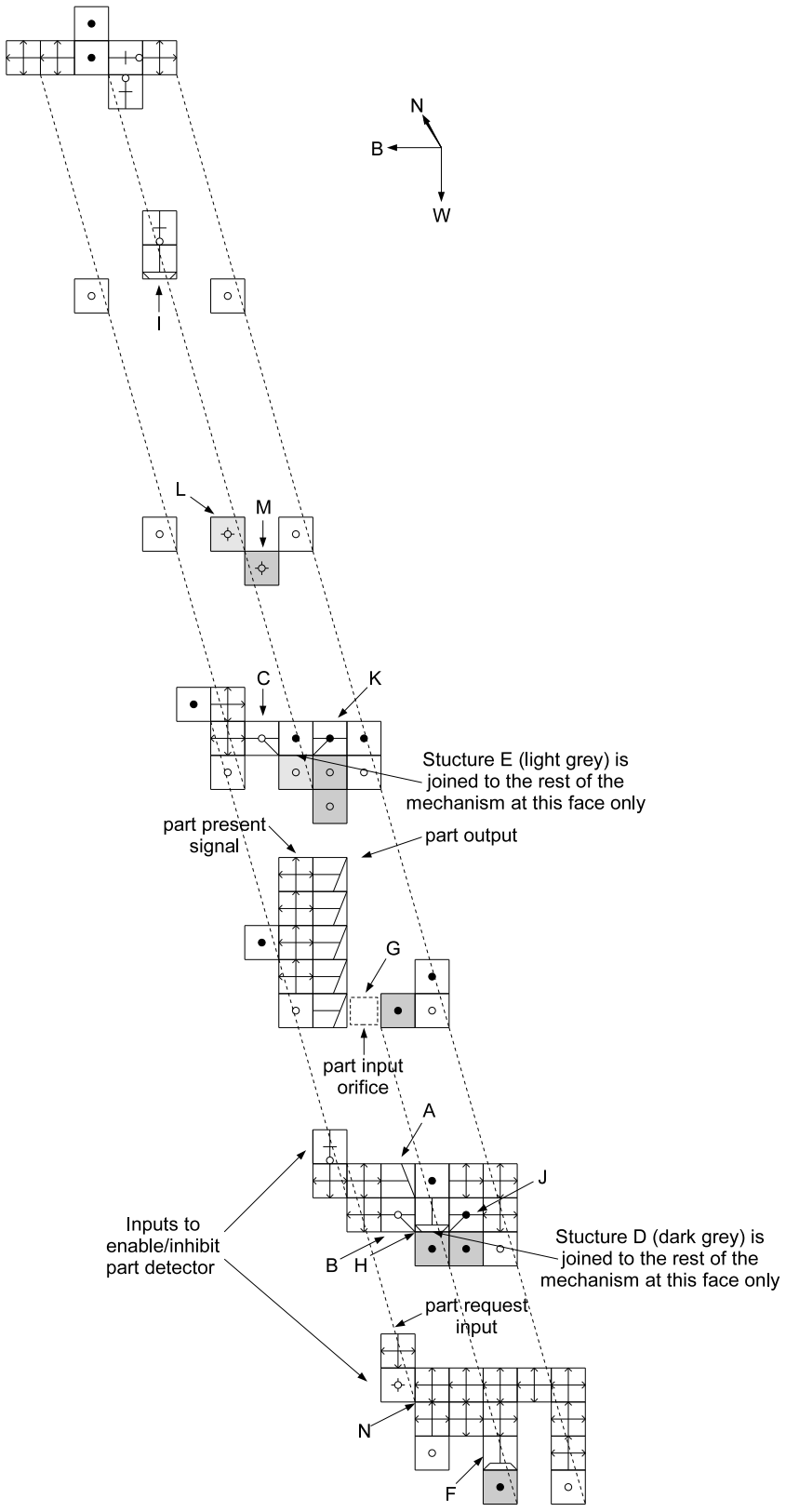


Figure 5.4. The *detect* mechanism.

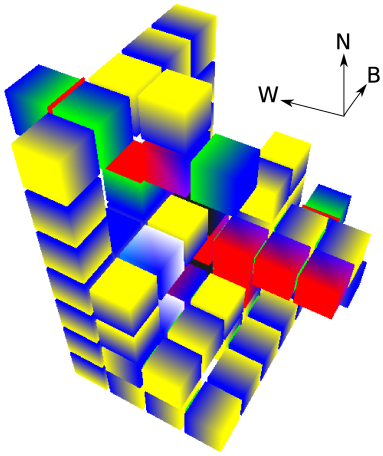


Figure 5.5. A graphical representation of the *detect* mechanism.

mechanisms.

A part enters the sorting mechanism from the *part-output* location of the *detect* mechanism. A 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.

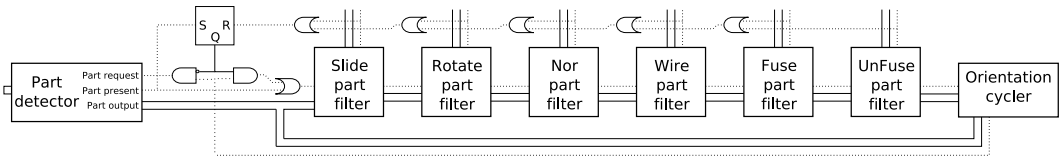


Figure 5.6. A schematic diagram of the *sorter* mechanism connected to the *detect* mechanism.

The following sections describe the modules of the *sorter* mechanism.

5.4.1 The orientation cycler

If a part is recognised by one of the six *filters* it will be diverted to a storage area connected to the $SOUTH$ of that *filter*. Any part not in a recognisable orientation will pass through all six *filters* without being diverted to a storage area. The part will then enter a mechanism called the *orientation cycler* that will alter the orientation of the part after which the part will be fed back to the first filter.

The purpose of the *orientation cycler* is to guarantee that any part in the *sorter* will eventually be recognised by cycling the part through all 24 possible orientations if

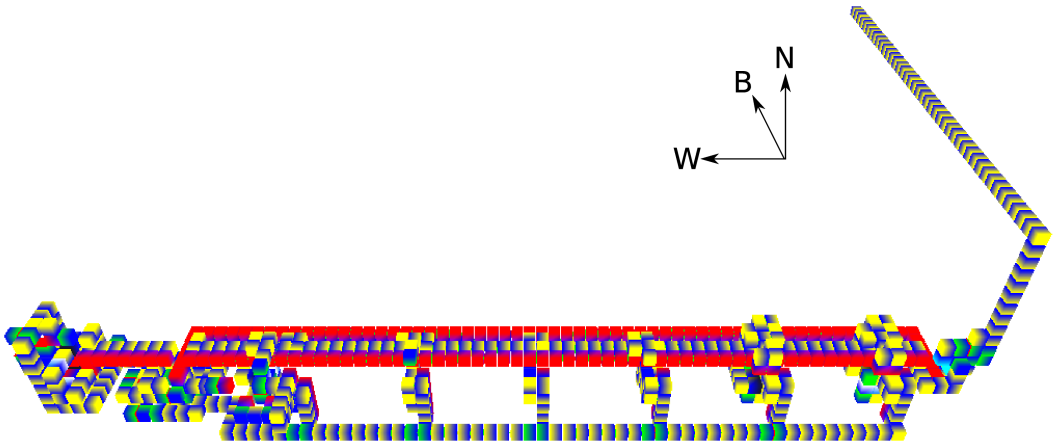


Figure 5.7. A graphical representation of the *sorter* connected to the *detect* mechanism.

necessary. The *orientation cycler* contains two *rotate* parts arranged as in Figure 5.8.

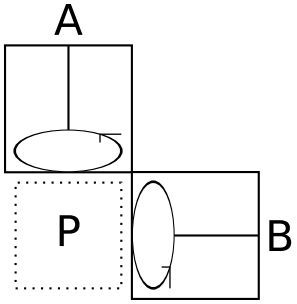


Figure 5.8. The arrangement of *rotate* parts in the *orientation cycler*.

If a part were to remain stationary at position P then that part could be cycled through all 24 orientations by applying at every time step a signal either to *rotate* part A or to *rotate* part B. Suppose that we have a sequence of Boolean values Q of length m that specifies at each time step whether a signal should be applied to A or to B. In the *orientation cycler* parts do not remain stationary at P but instead pass through several times, undergoing one change in orientation every pass. It takes T_{sorter} time units for an unrecognised part to traverse the *sorter* once. Therefore a part will enter the *orientation cycler* every T_{sorter} time units. In order to apply in order the sequence of rotations specified by Q to the part, we must arrange that during the n^{th} passage of the part through the *orientation cycler* the *rotate* part specified by $Q[(n-1) \bmod m]$ is active. One way to achieve this is to design a mechanism that advances through the sequence Q by one step every time a part

passes through the *orientation cycler*, looping back to the beginning when the end of Q is reached. Given that T_{sorter} is constant, a simple method of doing this is to use a signal loop containing a sequence Q' derived from Q as follows.

If the first passage of a part through the *orientation cycler* happens at time t_0 then the n^{th} passage happens at time $t_0 + nT_{sorter}$. Provided that T_{sorter} and m are coprime, then $(t_0 + nT_{sorter})$ mod m will take on all values 0 to $m-1$ in some order as n increments from from 0 to $m-1$. We can arrange a sequence $Q'[(t_0 + nT_{sorter}) \bmod m] = Q[n \bmod m]$ and then use Q' to activate either A or B as a part passes through the *orientation cycler*.

Q' is a sequence of Boolean values which will, when fed to the *orientation cycler* ensure that a part passing through the *orientation cycler* many times will eventually cycle through all possible orientations. Let us call any sequence with this property a cycling sequence.

Rather than have a signal loop dedicated to providing the cycling sequence Q' to the *orientation cycler* we can instead make use of the fact that if we choose any sequence of length l at random then the probability that it is a cycling sequence increases as l increases.

This probability for various values of l was calculated by sampling 1,000,000 random sequences for each of various lengths l. The results are shown in Table 5.1.

Sequence length l	Cycling sequence probability (2 d.p.)
25	0.00
50	0.01
75	0.49
100	0.78
125	0.92
150	0.97
175	0.99
200	1.00
225	1.00
250	1.00

Table 5.1. Probability that a randomly chosen sequence of length l is a cycling sequence.

Table 5.1 shows that for sequences of length 200 or more it is overwhelmingly likely that a randomly chosen sequence will be a cycling sequence.

The *memory* (described in section 5.8) contains a collection of 256-bit signal loops, and the layout of subsystems in the machine is such that running a signal path from the *memory* to the *orientation cycler* is very straightforward. The 256-bit sequence in the *memory* signal loop that is nearest to the *orientation cycler* is a cycling sequence.

Figure 5.9 shows the *orientation cycler*. A part p enters the mechanism at A at the

same time that a signal enters the mechanism at C. Input B is the input for the cycling sequence. A signal derived from C and B is used to activate *rotate* part D. A signal derived from C and not B is used to activate *rotate* part E. As p passes beneath D (and to the west of E) one of these *rotate* parts is active and rotates p. After this p emerges from the mechanism at F.

5.4.2 Nor filter and wire filter

Figure 5.10 shows the *nor filter*. This is the simplest of the six *filters*. A single part path leads into the *filter*, and two part paths lead out. The *filter* is capable of detecting a *nor* with its primary axis pointing $SOUTH$ (see axes in Figure 5.10). A part p travels into the *filter* along path A passing above the input to part B as it does so, and arrives at point C. Since a southward pointing *nor* part is the only possible part that can cause a signal to enter part B as it passes above it, a signal taken from B to *slide* part D will divert part p if and only if p is a southward pointing *nor* part. If p is any other part or is a *nor* part in any other orientation then it will exit the *filter* when part E slides it out of the *filter* (after having waited at C for long enough for any possible diversion to happen).

A *wire filter* can be constructed in a very similar way to the *nor filter* except that in this case a signal must be fed into the part being tested so that any possible output from it can be detected. Since a southward pointing *wire* part is the only part that will output a signal one time unit after receiving a signal on its input, a part from which such a signal is detected can be diverted and identified as a southward pointing *wire*.

5.4.3 Rotate filter

Figure 5.11 shows the *rotate filter*. A part p is fed into the *filter* at A at the same time that a signal is fed into the *filter* at B. p is transported to location C. As p passes beneath part D a signal is applied to the northward face of p by part D. If and only if p is a *rotate* part pointing southward will part E be rotated so as to allow a signal derived from B to pass through E, and ultimately to activate F and divert p to H. Otherwise p will emerge from the filter as it moves $EAST$ from location C.

5.4.4 Slide filter

The *slide filter* is the only *filter* that can recognise a part in a number of functionally distinct orientations. It can recognise a *slide* part in any of the orientations in which its primary axis points $SOUTH$ regardless of the orientation of the part's secondary axis.

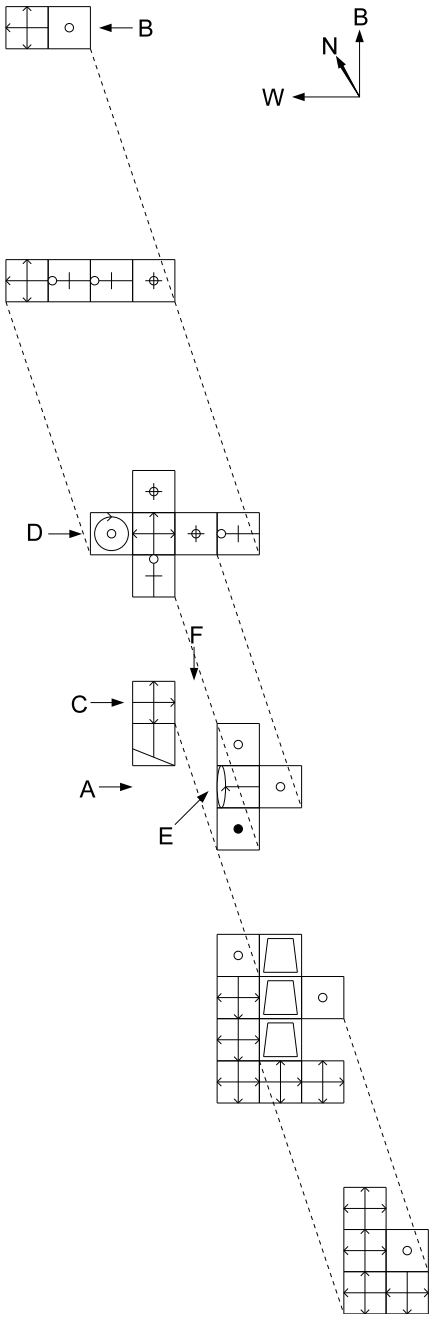


Figure 5.9. The *orientation cycler*.

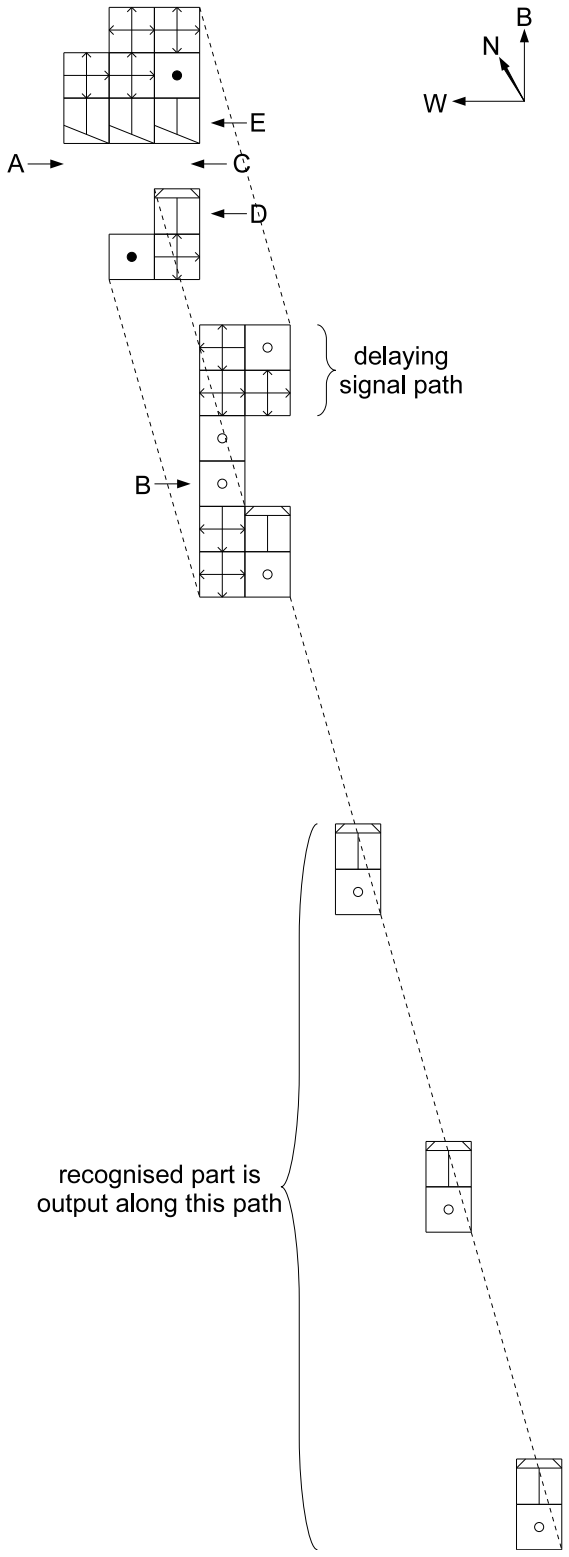


Figure 5.10. The *nor filter*.

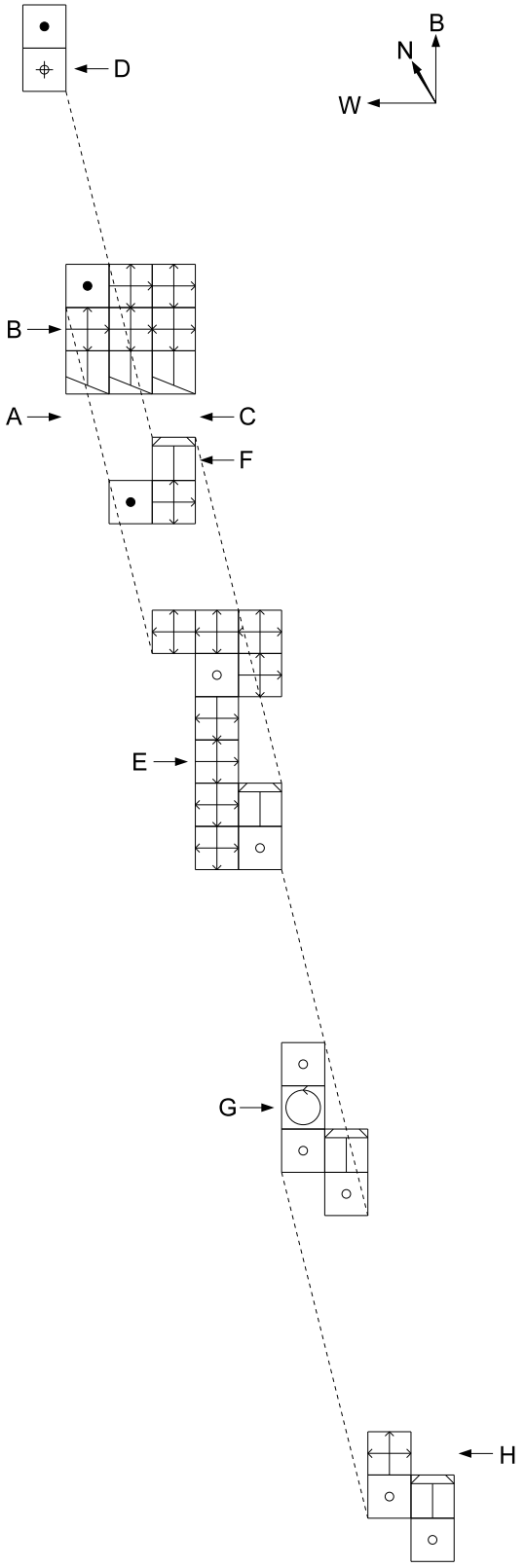


Figure 5.11. The *rotate filter*.

This is necessary because all of the *filters* except the *nor filter* work by applying a signal to the $NORTH$ face of a part. The *slide filter* is the first *filter* in the series of *filters* that a part passes through. If any southward pointing *slide* part were to enter any of the other *filters*, the result would be that when a signal was applied to the $NORTH$ face of the *slide* part then the part immediately beneath the *slide* part and any parts connected to it would be moved. This would lead to the whole machine moving in an unpredictable way.

Figure 5.12 shows the *slide filter*. A part p enters the filter at A and moves to location B where a signal is applied to the $NORTH$ face of the part by part C. If p is a *slide* part pointing southward, structure D will move $EAST, WEST, BACK$ or $FRONT$ and immediately be moved back to its original position by the nearby *slide* parts. This will cause *nor* part E to inject a signal into mechanism H at a point that will cause p (which has by now moved to location G) to adopt the SF orientation regardless of its original orientation. The movement of D will also cause a pulse to emerge from *nor* part F which will then cause p to be diverted to emerge at I. If p is not recognised then it will emerge at J.

5.4.5 Fuse filter and unfuse filter

Figure 5.13 shows the *fuse filter*. A part p is fed into the *filter* at A at the same time that a signal enters at B. p is transported to location C. A signal derived from B is applied to *unfuse* parts E so that there is no connection at joints F. After this a signal derived from B is applied to the $NORTH$ face of p by part D. If and only if p is a *fuse* part in an SF orientation will the frontmost of the joints F become connected. *Slide* parts I will slide the structure beneath them back and forth, as a result of which the signal output by *nor* part G will arrive at H, but only if p was a SF orientated *fuse* part. A signal derived from H can then be used to divert p along J, otherwise p will exit from the *filter* at L. *Fuse* parts K ensure that joints F are connected again once the *filter* has finished the test on p.

Figure 5.14 shows the *unfuse filter*. Its operation is the same as the *fuse filter*, except for the following: *Unfuse* part A disconnects the joint B. If the part being tested is a SF orientated *unfuse* part then joint C will be disconnected, causing a signal to arrive at D when slide parts E are activated.

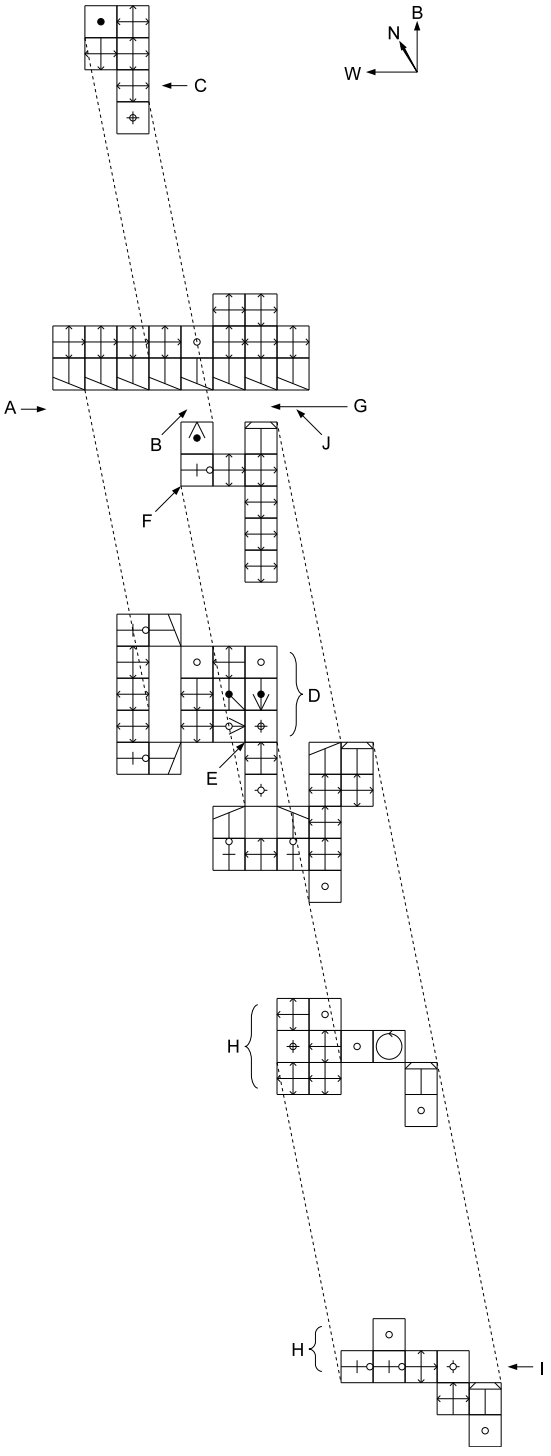


Figure 5.12. The *slide filter*.

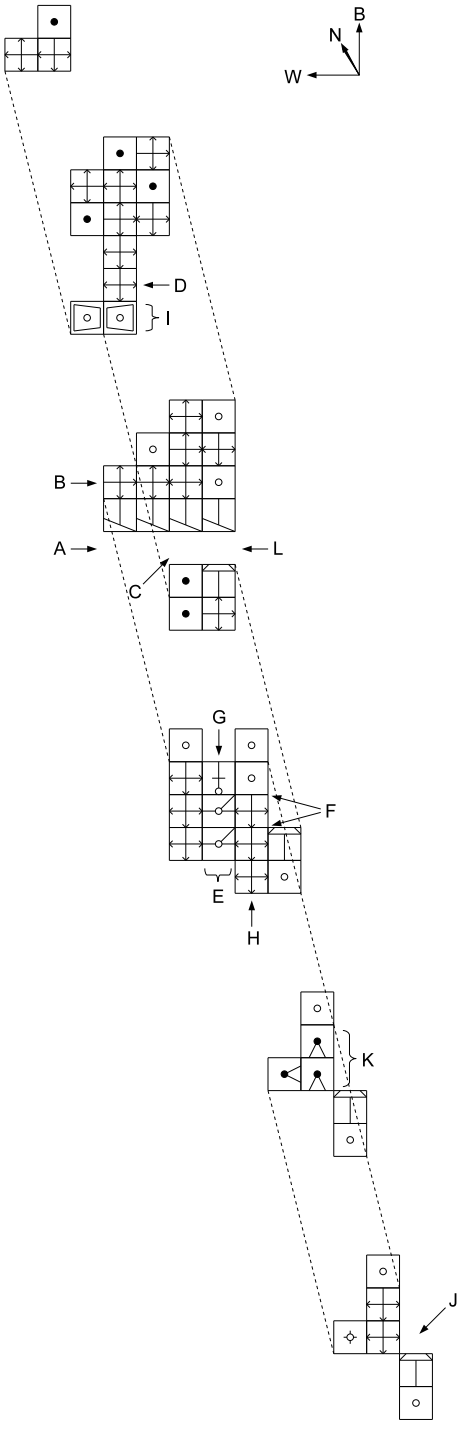


Figure 5.13. The *fuse filter*.

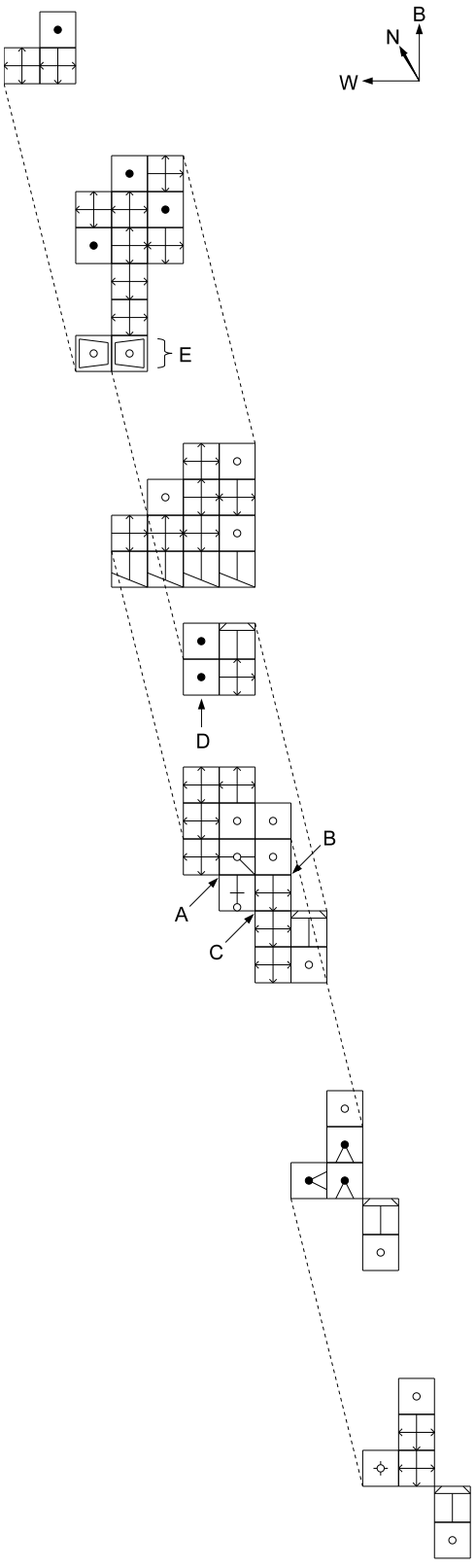


Figure 5.14. The *unfuse filter*.

5.5 Part storage and dispensing

There is a separate *storage mechanism* for each type of part. Each *storage mechanism* is based around a two dimensional rectangle of parts. Figure 5.15 is a schematic diagram of a single *storage mechanism*. Figure 5.16 shows a part-level diagram of the mechanism and Figure 5.17 shows a graphical representation of the mechanism.

Parts shaded in the same colour in Figure 5.16 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 to which it is connected via the $WEST$ face of the *wire* part $SOUTH$ of *unfuse* part G.

The *storage dispenser* mechanism is not connected to any other mechanism except the rectangle 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.

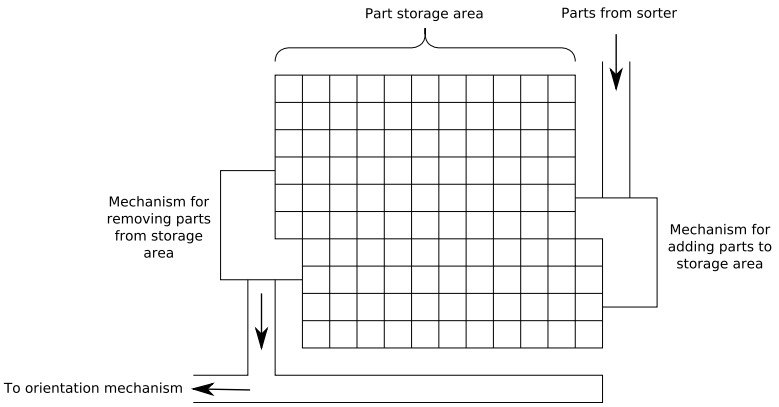


Figure 5.15. A schematic diagram of the *storage mechanism* for a single type of part.

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. A part-level diagram of the mechanism for adding parts onto the rectangle is shown towards the bottom right hand corner of Figure 5.16. This mechanism will be referred to as 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 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 $NORTH$ 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 p coming into the full *storage mechanism* from A will pass all the way $SOUTH$ to *slide* part Q, which will then move p out of the *storage mechanism* along path R and return it to the environment.

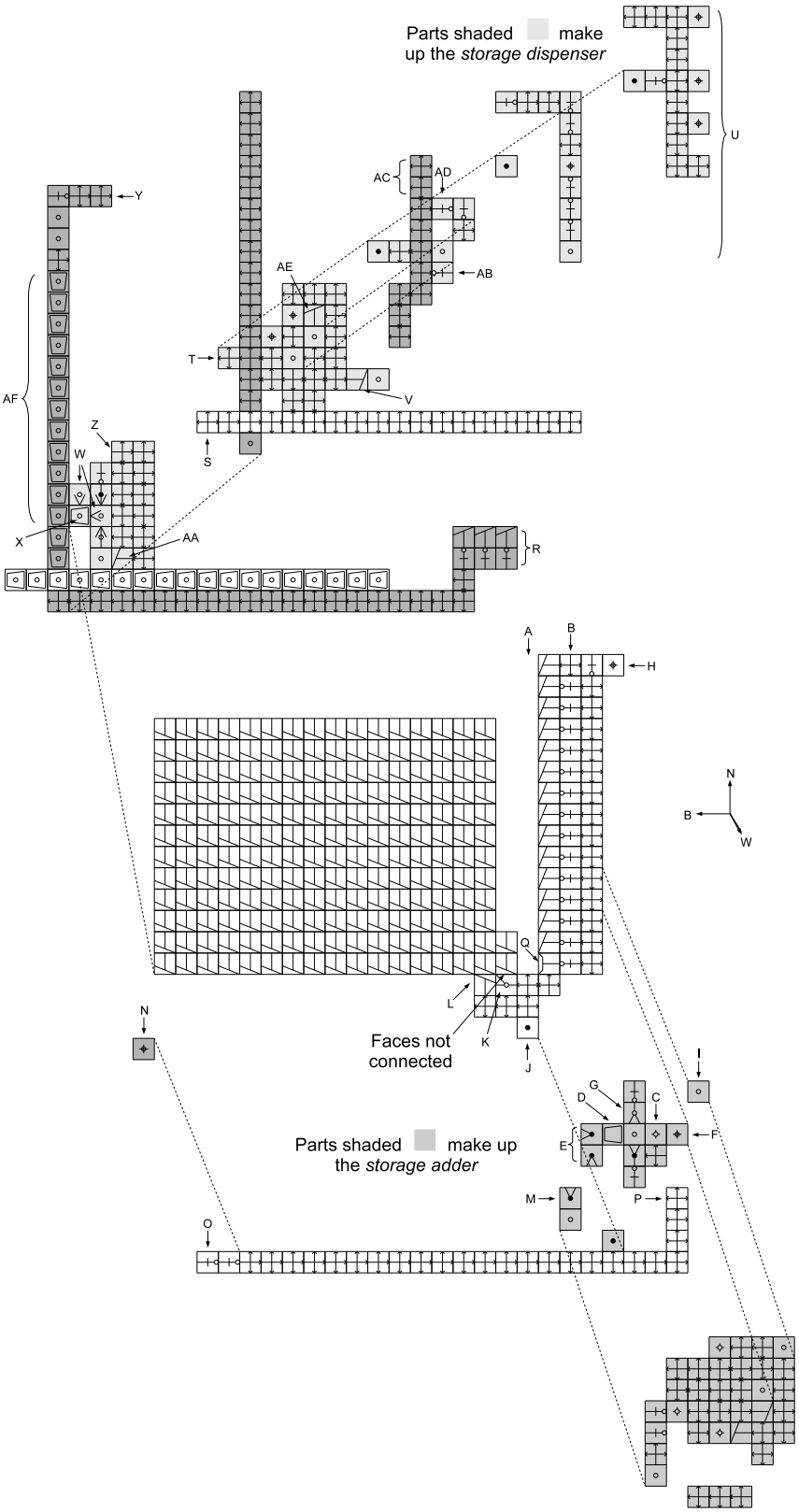


Figure 5.16. The *storage mechanism* for *slide* parts.

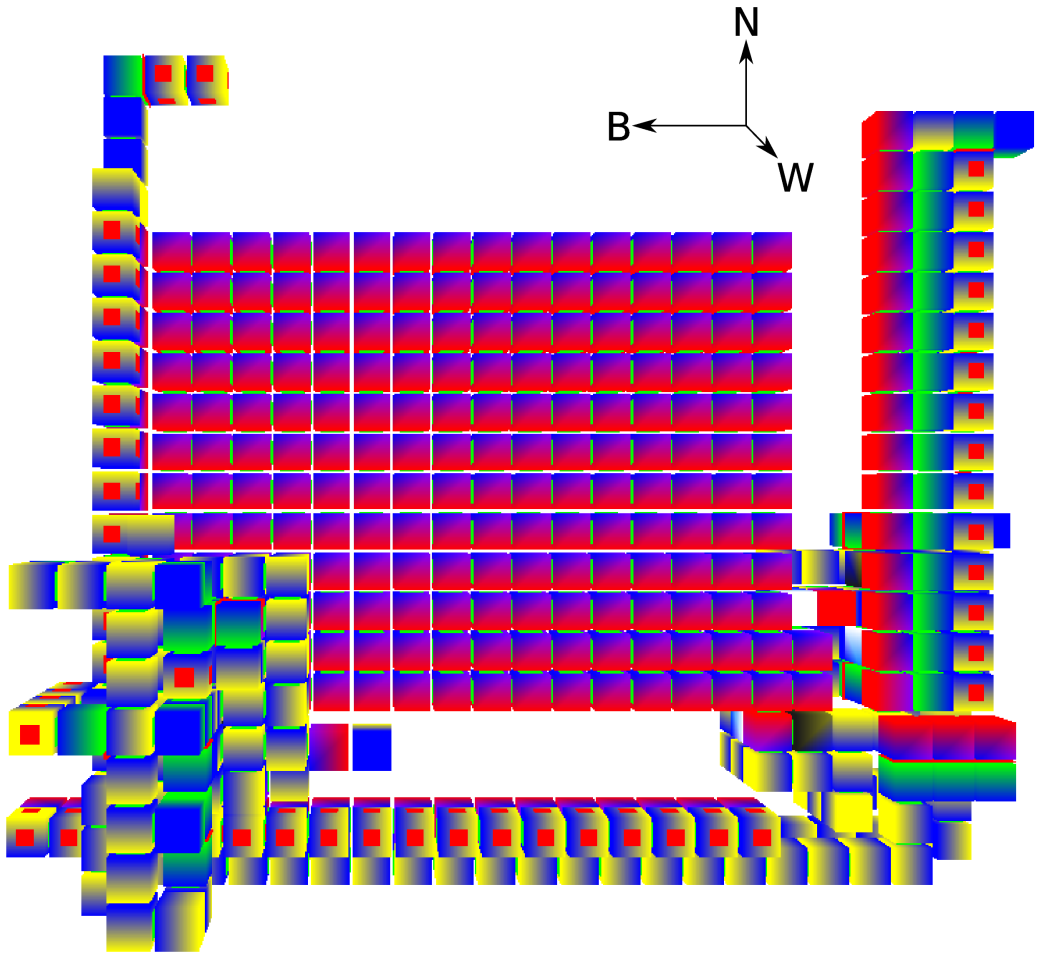


Figure 5.17. A graphical representation of the *storage mechanism*.

When the machine's control unit requests a part from a *storage* mechanism the part is retrieved from the $BACK$ edge of the rectangle. 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 Figure 5.16.

A serial 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 in the rectangle and *slide* part W 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 5.2 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.

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

Table 5.2. Serial encoding used for dispensing parts.

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 with each other's operation. For this reason the effective capacity of a *storage mechanism* is 180 parts. This means that any construction operation that the machine performs between one round of part collection and the next must not use more than 180 parts of any single type.

5.6 Orientation

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 which orientation a part should be in.

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 from the register be fed to some collection of *wire*, *nor* and *rotate* parts 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. It does not matter to us which word results in which orientation.

The simplest possible arrangement of this type is to have a part path running, say, from $FRONT$ to $BACK$ and then connect each bit from the register to a *rotate* part that will operate on parts passing along this path. Clearly some of these *rotate* parts will need to rotate about different axes from others in order to span the whole range of possible rotations. We will also need to ensure that no attempt is made to move a part at the same time that it is being rotated.

Figure 5.18 shows a mechanism of this type that was found to exhibit the correct behaviour. Figure 5.19 shows a schematic diagram of this mechanism. In Figure 5.19, dashed lines represent signal paths using the *pulse* signal representation and solid lines represent signal paths using the *static* signal representation. Boxes containing the letter D represent delays of 2 time units. Figure 5.20 shows a graphical representation of the *orientation* mechanism.

For each possible value of the orientation register Table 5.3 shows what the resulting orientation is for parts emerging from *part-output* of the *orientation* mechanism.

5.7 Construction arm

Section 4.3.7 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 similar to those described in section 4.3.8 are used to decode a sequence 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 the

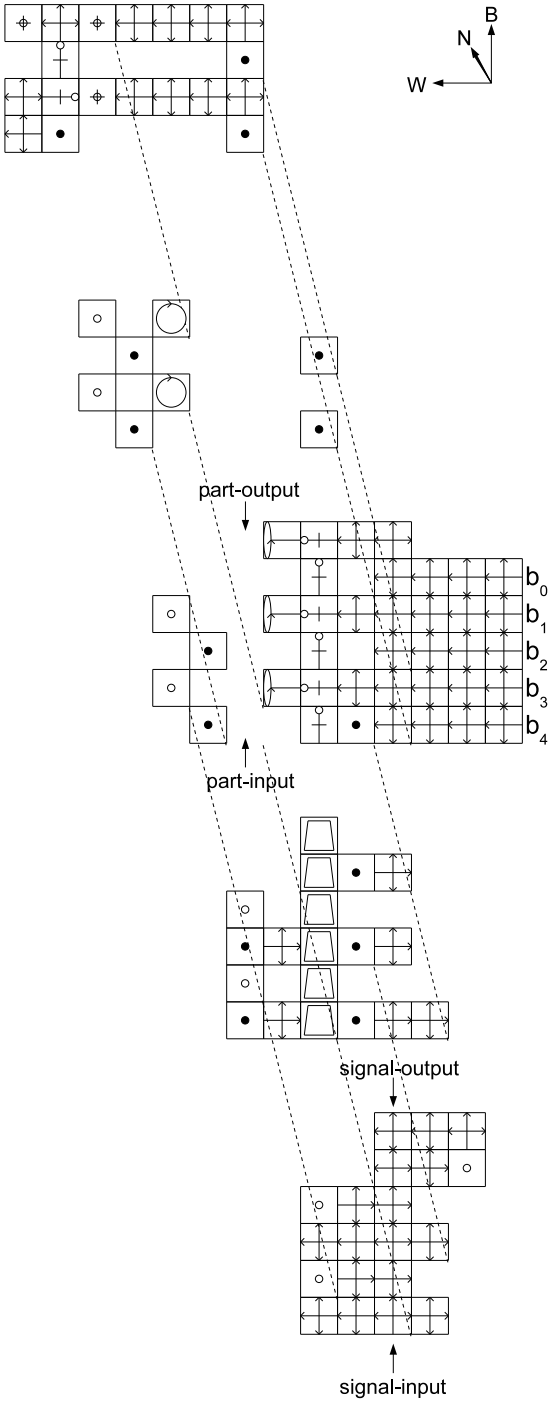


Figure 5.18. The *orientation* mechanism.

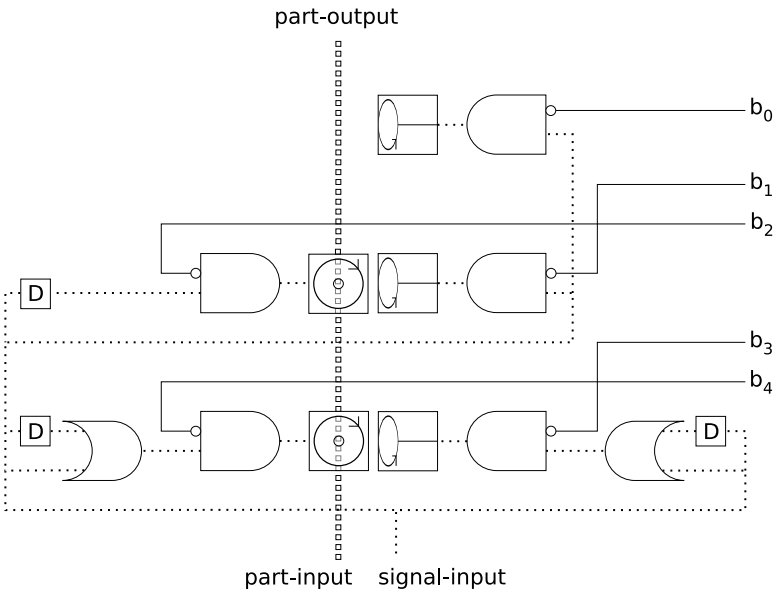


Figure 5.19. A schematic diagram of the *orientation* mechanism.

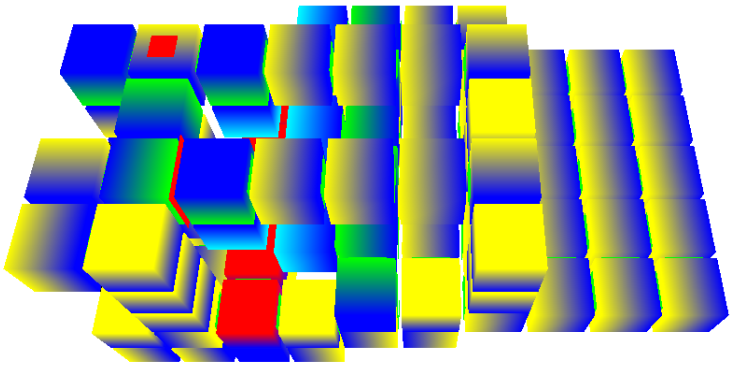


Figure 5.20. A graphical representation of the *orientation* mechanism.

different segments of the construction arm to move back and forth along one axis. Table 5.4 shows the encoding used for each movement of the *construction arm*.

Figure 5.21 shows a single joint between two signal paths in the *construction arm*. Figure 5.22 shows a schematic diagram of the *construction arm*. Figure 5.23 shows a graphical representation.

The operation of the joint shown in Figure 5.21 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 signal path at right angles to this one.

Value	Orientation	Value	Orientation
00000	EB	10000	EF
00001	EN	10001	ES
00010	BW	10010	BE
00011	NW	10011	NE
00100	SB	10100	SF
00101	BN	10101	BS
00110	BN	10110	BS
00111	NF	10111	NB
01000	WF	11000	WB
01001	WS	11001	WN
01010	FE	11010	FW
01011	SE	11011	SW
01100	NF	11100	NB
01101	FS	11101	FN
01110	FS	11110	FN
01111	SB	11111	SF

Table 5.3. Orientation of parts at *part-output* for different values of the orientation register.

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

Table 5.4. Serial encoding used for *construction arm* movements.

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 one bit (see Table 5.4). 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.

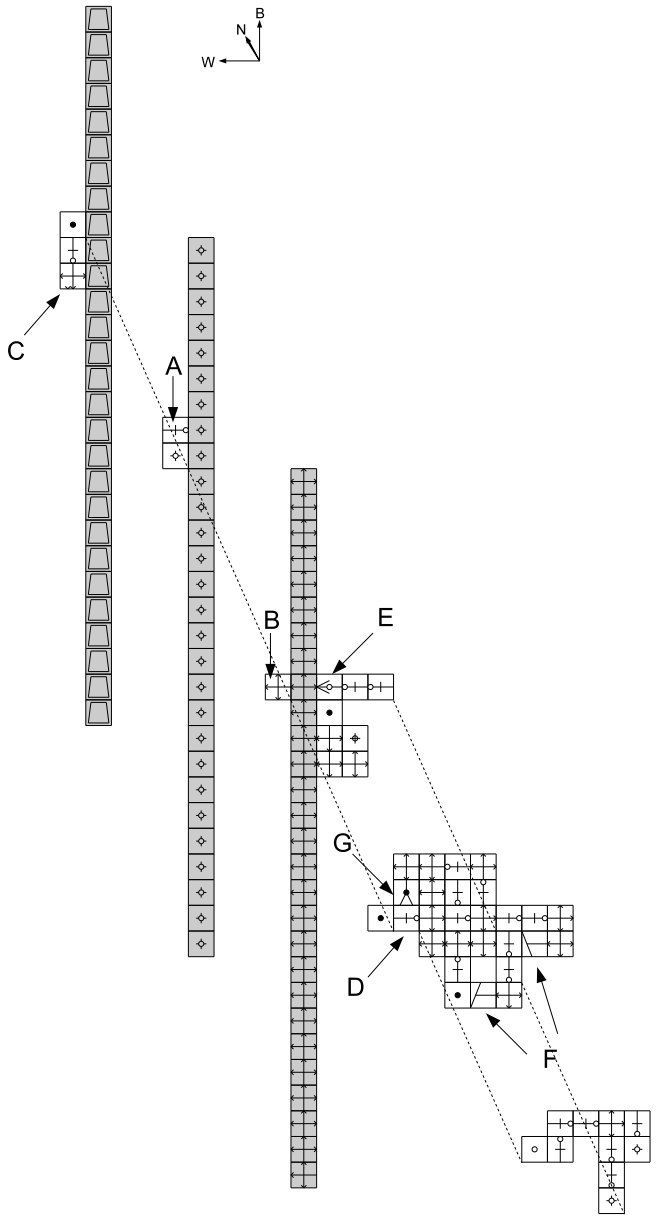


Figure 5.21. A single joint between two paths in the *construction arm*.

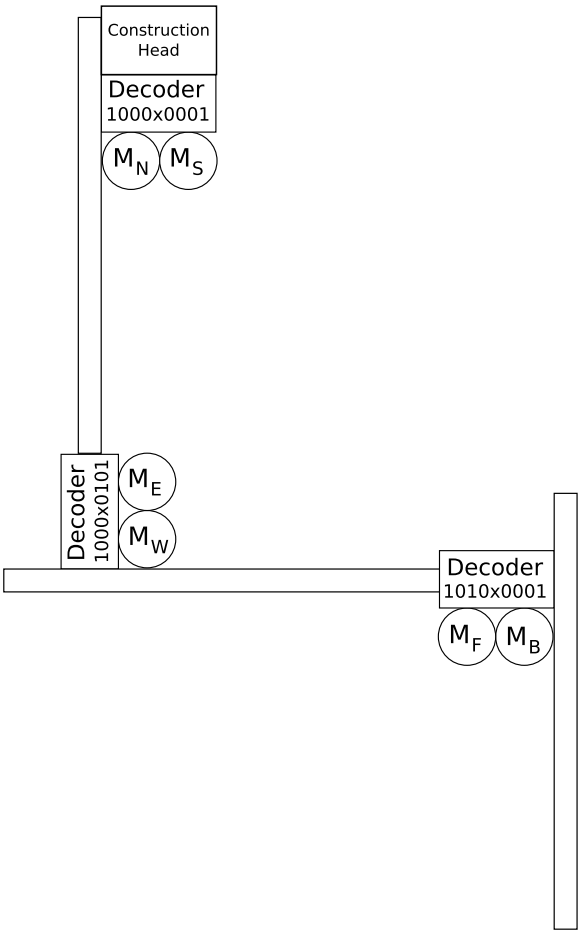


Figure 5.22. A schematic diagram of the *construction arm*.

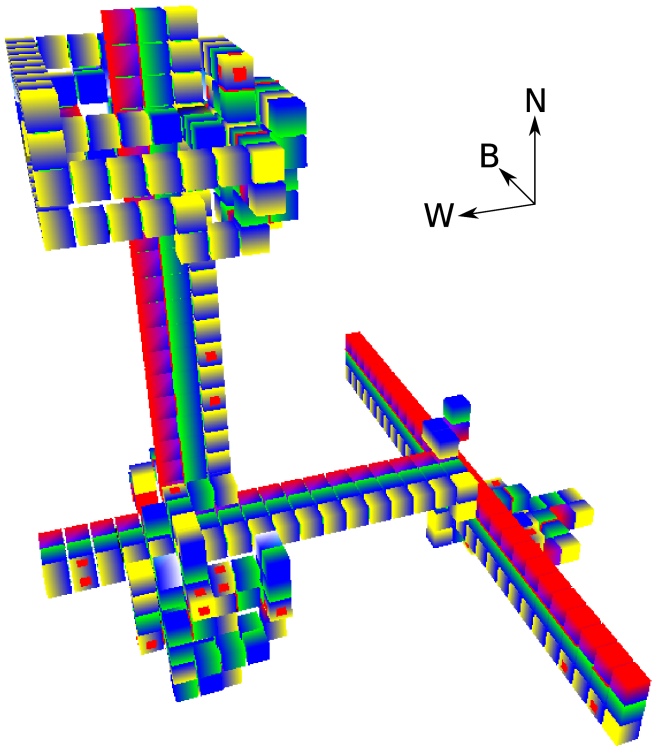


Figure 5.23. A graphical representation of the *construction arm*.

At the very end of the *construction arm* is a mechanism called the *construction head*, shown in Figure 5.24. This contains decoders and *slide* parts that fulfill the same function as in a joint. The *fuse* parts labelled F ensure that newly placed parts are connected to the structure being constructed. The *construction 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 move the *construction head* $NORTH$. 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.

5.8 Memory

Having designed the *construction arm*, *part dispenser* and *orientation* mechanism, it is now possible to devise an instruction set. The arrangement of the memory and the design of the instruction set go hand-in-hand: each memory word must be large enough to contain an instruction. The arrangement of the memory is presented in this section and the instruction set architecture is presented in section 5.10.

Possible methods of implementing a memory in CBlocks3D were described in section 4.3.10. We choose the method of using gated signal loops as described in section 4.3.4 because it simplifies the problem of copying the contents of one memory unit to another when the contents of the parent machine'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 fact that it is possible to copy the memory contents from parent to child in this simple way is a great advantage. In effect this process is the equivalent of the Tape

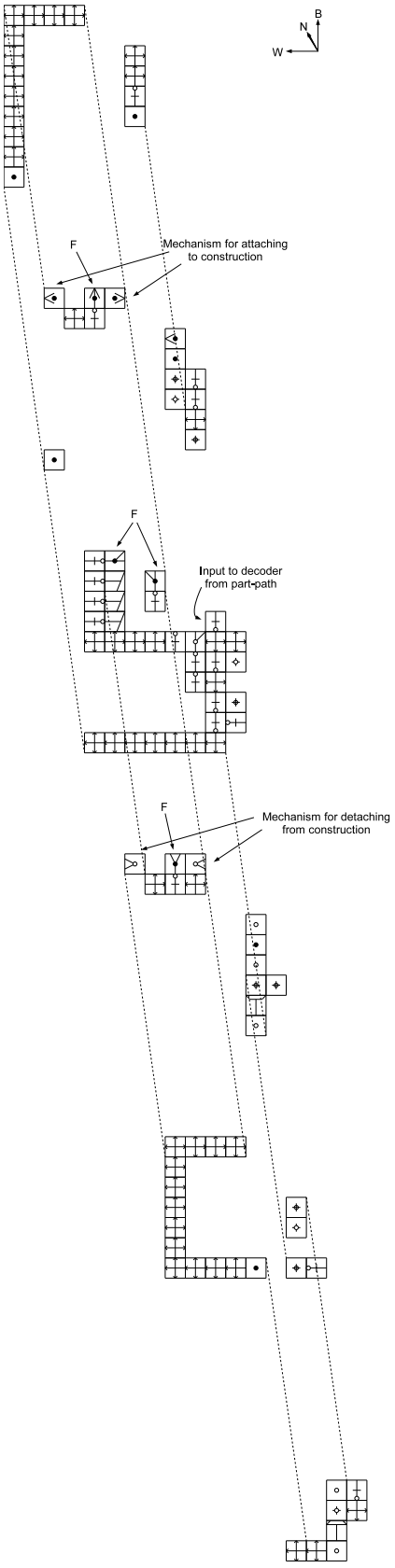


Figure 5.24. The *construction head.*

Copying Logic in von Neumann's architecture (see Figure 2.6).

The signal loop used as the basis for the machine'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 5.25 shows how each memory loop is arranged.

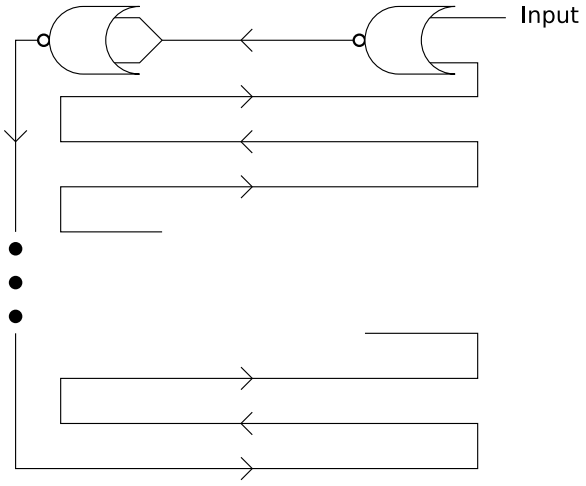


Figure 5.25. The arrangement of the memory signal loop.

Each 256-word memory module is made using 6 gated signal loops placed side-by-side from $WEST$ to $EAST$. The machine contains 32 memory modules giving a total of 8,192 6-bit words or 49,142 bits. Each module has a 6-bit wide AND gate connected to its output. Figure 5.26 shows this gating mechanism. The output 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.

As can be seen from the diagram of the 6-bit AND gate mechanism in Figure 5.26 the output from each memory loop is offset 1 bit frontward from that of the previous loop. Also each memory module is 6 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.

The reason for offsetting the output of each signal loop in a given memory module by 1 bit is to simplify the structure of the instruction decoders (see section 5.10). The outputs from each loop are combined through various gates in order to decode instructions, and

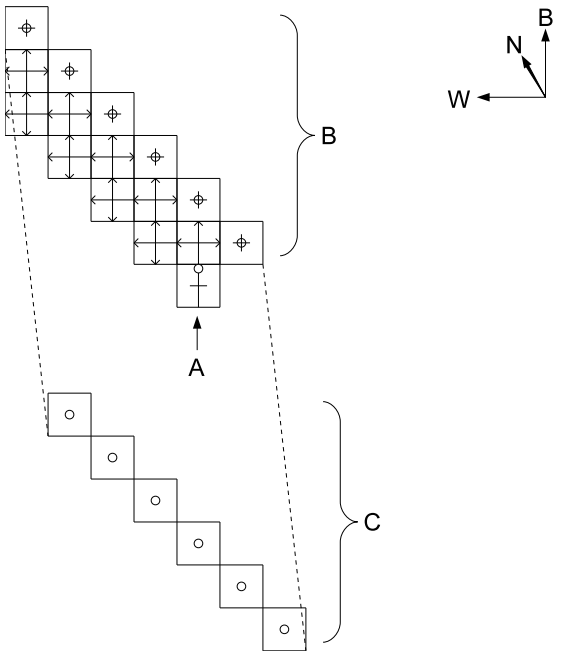


Figure 5.26. The 6-bit AND-gate connected to each memory module.

we must ensure that any cell which combines signals originating from particular memory outputs is equidistant from each output. This can be done in a compact area if the outputs are arranged diagonally.

5.9 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 5 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.

It would be possible to design the address decoder using only *wire* and *nor* parts, but the length of time that it would take for the effect of the inputs to propagate to the output of the 32^{nd} address decoder would be at least 192 time units. It would take at least a further 192 time units for the output of the 32^{nd} memory module to reach the memory output. This is longer than the minimum possible time that could occur between successive outputs from the *comparator* that compares the lower 8 bits of the *program*

counter with the *memory address counter*. Therefore the correct memory word would not have arrived at the output of the memory unit by the time that it was required.

A way of avoiding the propagation delays associated with using *wire* and *nor* parts is to make use of the kinematic behaviour of the CBlocks3D environment and use moveable rods to propagate signals over long distances. For the address decoder this is particularly straightforward. It also requires fewer parts and has a simpler structure than an address decoder made from *wire* and *nor* parts.

The address decoder consists of 5 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. As shown in Figure 5.27 each rod is made from $NORTH$ facing *wire* parts except for 32 special parts call path-forming parts which 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 position for the 5 rods, exactly one signal path is formed from the alignment of path-forming parts so 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:

$p_0(n) = 1$ if $n \bmod 2 > 0$, $p_0(n) = 0$ otherwise

$p_1(n) = 1$ if $n \bmod 4 > 1$, $p_1(n) = 0$ otherwise

$p_2(n) = 1$ if $n \bmod 8 > 3$, $p_2(n) = 0$ otherwise

$p_3(n) = 1$ if $n \bmod 16 > 7$, $p_3(n) = 0$ otherwise

$p_4(n) = 1$ if $n \bmod 32 > 15$, $p_4(n) = 0$ otherwise

A mechanism is required to convert the output of the upper 5 bits of the program counter (which use the static signal representation) into an offset for each address decoder rod. Figure 5.28 shows a single bit of this mechanism called the *selector* mechanism. If a

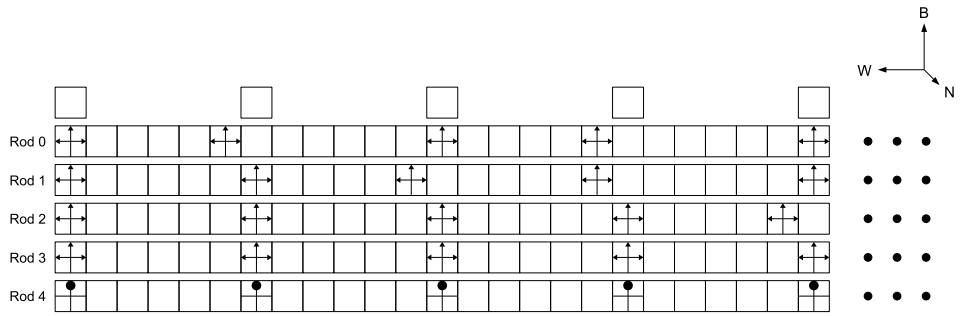


Figure 5.27. The structure of the address decoder.

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.

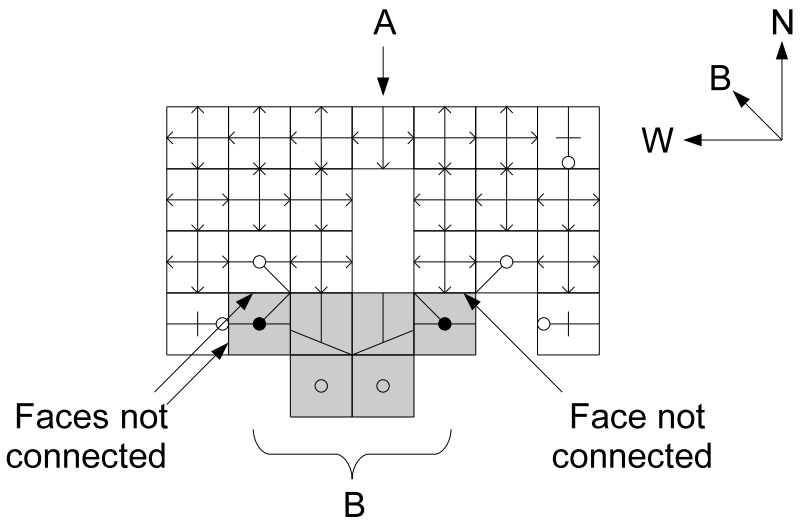


Figure 5.28. The *selector* mechanism for converting a static signal value to an offset.

5.10 Instruction set

The instruction set must contain instructions for carrying out the following operations:

- Moving the construction head in each of 6 directions
- Specifying 24 orientations
- 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 machine requires some mechanism for repeating sequences of instructions. In order for the memory to be able to store the instruction sequence required for replicating the entire machine, the number of instructions required for constructing the memory must fit into the memory itself. This is only possible if the memory has a regular structure that can be described by a repeated sequence of instructions. By enabling instruction sequences to

be repeated we will also reduce the number of instructions needed for constructing other parts of the machine that also have redundancy or regularity in their structure.

Two possible mechanisms for repeating 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, it was decided to implement a subroutine call mechanism but not a loop mechanism.

Table 5.5 shows the complete instruction set for the machine along with the encoding chosen for each instruction.

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	1ppppp	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 $4096 + 4a$
RETURN	011000	Return from the current subroutine
GATHER	010000	Begin collecting parts.
NOP	000000	No operation - do nothing

Table 5.5. Instruction encoding scheme.

A CALL instruction requires two instruction words and contains a 10-bit operand a. The destination address for a call instruction is $4096+4a$. 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 4 word boundary. This results in some wastage of memory when the length of a subroutine is not a multiple of 4 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.

This instruction encoding used for movement of the *construction arm* and for dispensing parts is chosen in such a way as to lead to an efficient design for the mechanism that interprets instruction words: the *instruction decoder*. Both instruction decoding and serial encoding for movement and dispensing instructions are combined within the *instruction decoder*. Figure 5.29 shows this mechanism. The mechanism also contains decoders for the RETURN and GATHER instructions.

In Figure 5.29 the diagonal line of *wire* parts labelled A is the 7-bit input to the decoder. The first 6-bits come directly from the memory. The 7^{th} bit is derived from the signal from the *comparator* that is used to compare the lower 8 bits of the *program counter* with the *memory address counter*. The 7^{th} bit tells the *instruction decoder* that there is something to decode.

The decoding and interpretation of CALL and ORIENT instructions is carried out by the control unit.

5.11 Control unit

The *control unit* is the part of the machine 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 the correct interpretation of CALL and ORIENT instructions.

Figure 5.30 shows a schematic diagram of the *control unit*.

5.11.1 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

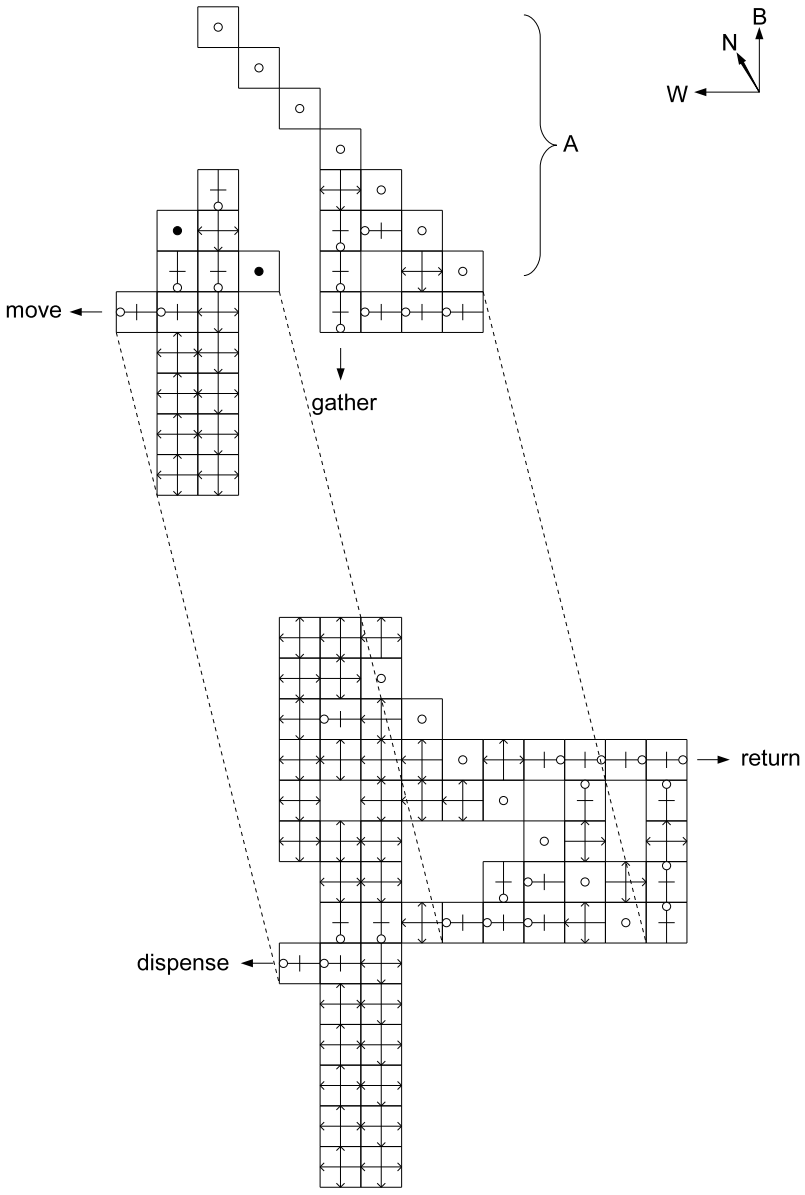


Figure 5.29. Instruction decoder.

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 multiplied by 4 and loaded into bits 2 to 11 of the *program counter*. Bits 0 and 1 of the *program counter* are set to 0 and

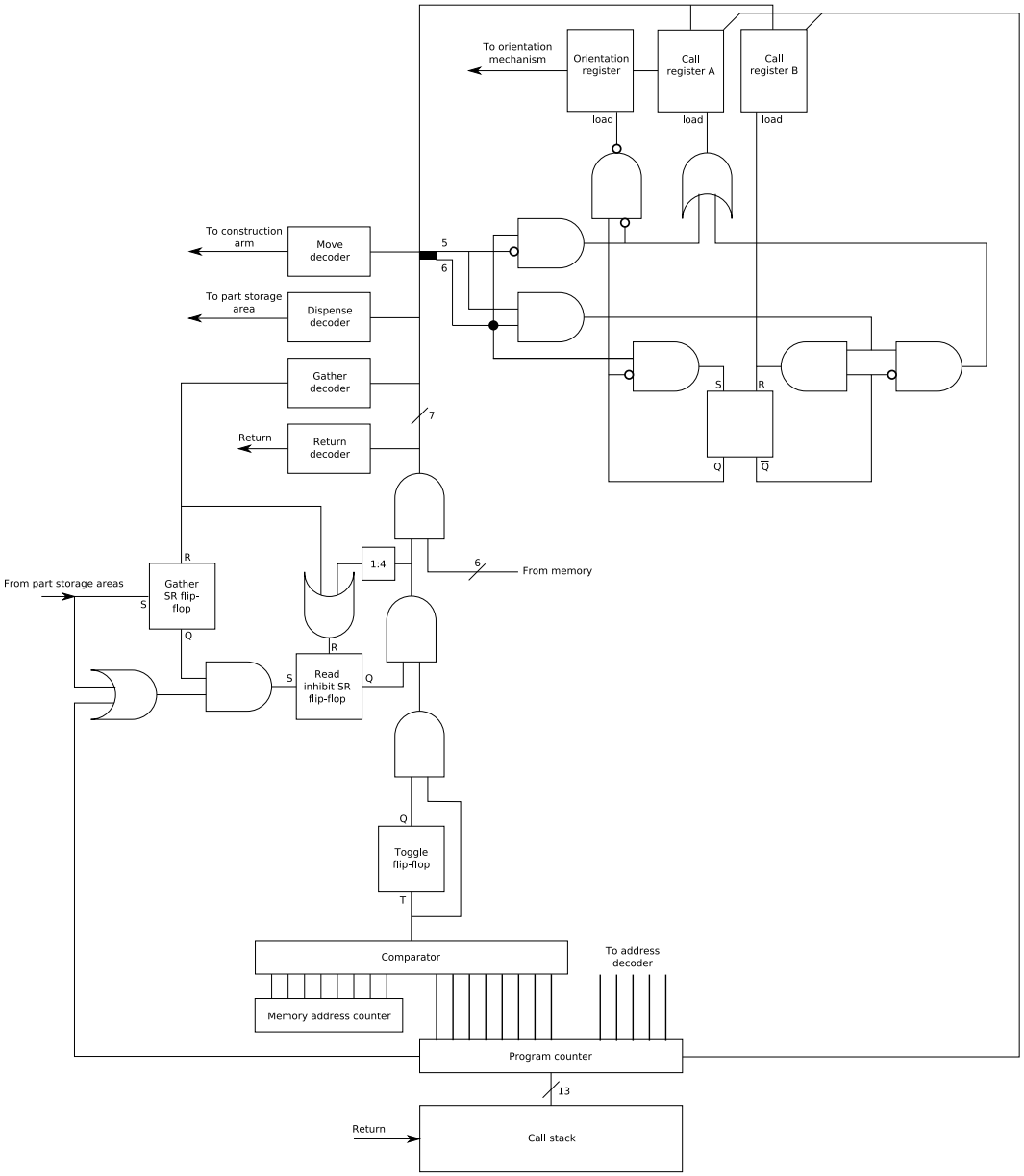


Figure 5.30. The control unit.

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 issued 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*. After a value is pushed onto the *call stack*, the whole array is moved backwards. Before popping a value from the *call stack*, the whole array is moved forward.

Figure 5.31 shows a single bit of the *program counter* and the interface between the *program counter* and the *call stack*.

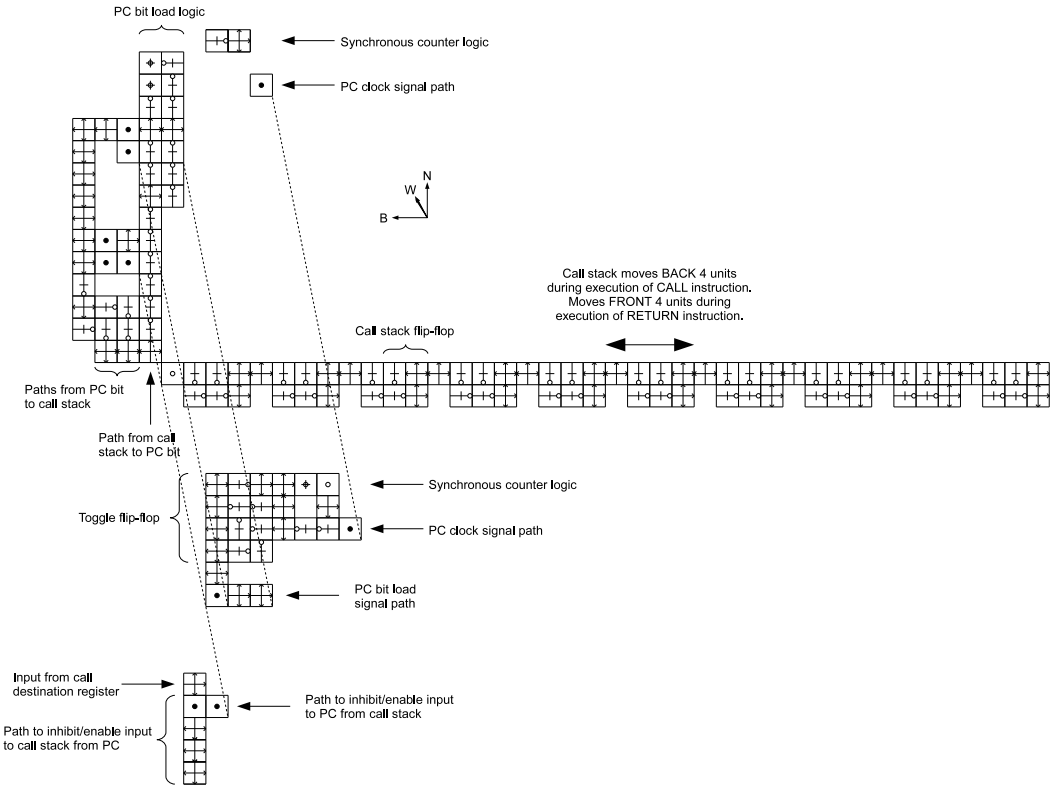


Figure 5.31. A single bit of the *program counter* and *call stack*.

Figure 5.32 shows a graphical representation of the program counter and call stack.

5.11.2 Memory address counter and comparator

The word being output by the memory unit at a point in time depends upon which of the 32 memory modules is selected and the time t. If we define t_0 as the time at which memory module 0 outputs word 0 when memory module 0 is selected then the word being output at time t when memory module m is selected is given by $((t - t_0) \bmod 256)) + 256m$.

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

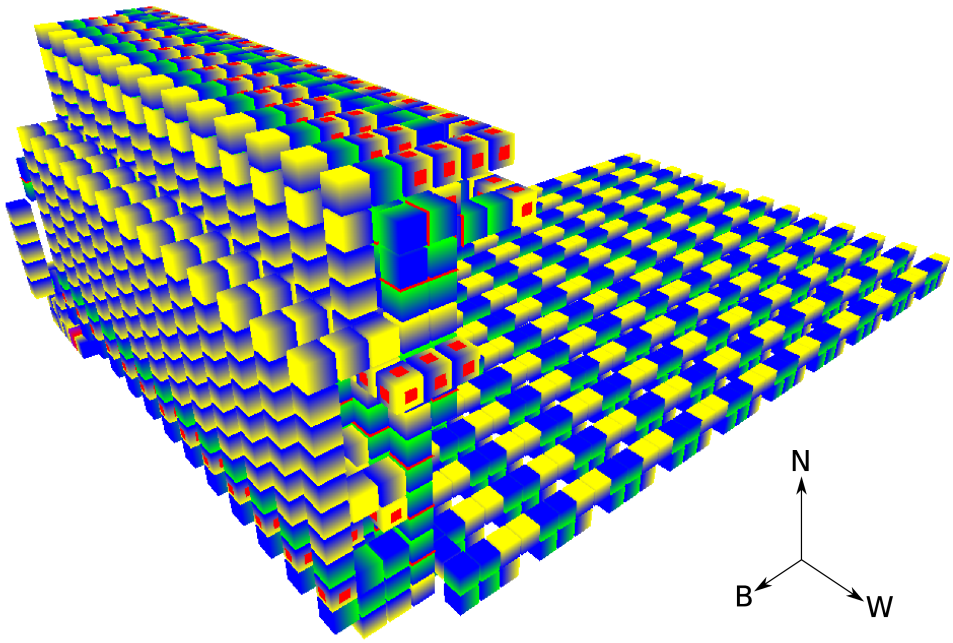


Figure 5.32. A graphical representation of the program counter and call stack.

module of the word being output matches the lower 8 bits of the *program counter*. In order to do this comparison we need 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 unit. This counter is called the *memory address counter* and is shown in Figure 5.33. Figure 5.34 shows a graphical representation. 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.

Figure 5.35 shows an 8-bit comparator. A graphical representation is given in Figure 5.36. The *comparator* compares the values of 8-bit *pulse* signal inputs A and B to produce a *pulse* signal output O. If all 8-bits of input A match those of input B then a logic 1 *pulse* signal will emerge at O 18 time units later, otherwise a logic 0 *pulse* signal will emerge.

When the *program counter* is being updated as a result of an increment from one address to the next or as a result of a CALL or RETURN instruction, the outputs of the *program counter* are indeterminate for a short period of time and therefore the output of the *comparator* may not be meaningful. The *read inhibit flip flop* shown in Figure 5.30 fulfills this role of preventing the output of the *comparator* from reaching the *instruction decoder* during the period of time when the *program counter* updates.

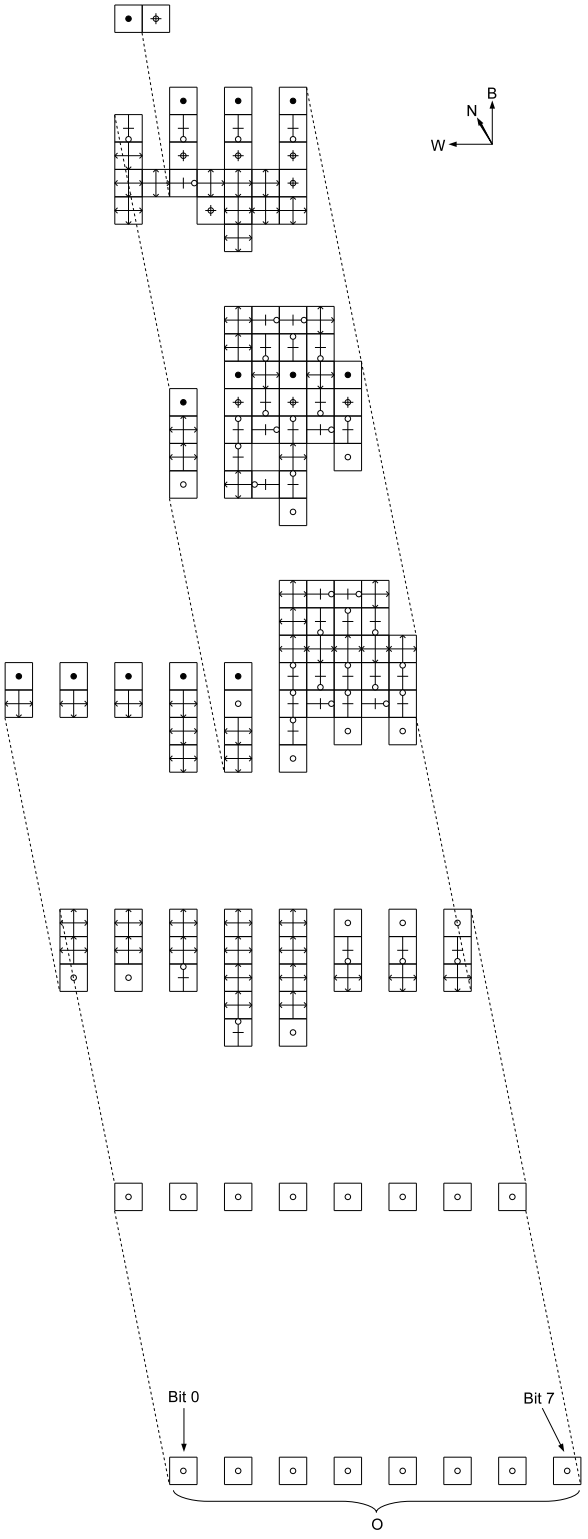


Figure 5.33. The *memory address counter*.

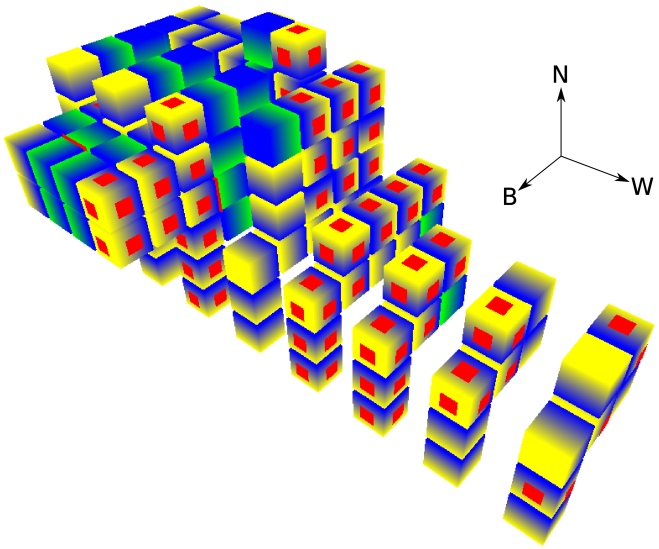


Figure 5.34. A graphical representation of the *memory address counter*.

5.11.3 Call and orientation registers

As described in section 5.10 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 the specify orientation of parts.

Figure 5.30 shows the logic required to interpret CALL and ORIENT instructions correctly. 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 second 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 of the second instruction word is set, then the 5 least significant bits of this word are stored in *call register B*, and a *4-pulse* 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 *pulse* signal inputs 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 5.37 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 logic 0 and put the register into a state where future logic 1 inputs to the parts labelled A will cause individual bits to be set. After a logic 1 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 effectively 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 5.38 shows a graphical representation of the call register.

The *orientation register* takes its *static* signal inputs from *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 Figure 4.6.

5.12 Programming the SRPC

Construction programs for the machine are written using the instruction set described in section 5.10. Programs are assembled using a simple single pass assembler and then written into memory modules using a function written for this purpose as part of the suite of classes described in section 4.4.2.

The construction program required for self-replication fits into 8,191 words. The fact that this is close to the capacity of the memory is no coincidence. During the development of the machine it was difficult to estimate the final size of the construction program. After the machine was designed and the construction program was written it was found to be several hundred words larger than the capacity of the memory. The program was examined very carefully to look for every opportunity to use subroutines to reduce the program size below 8,192 words.

One method of reducing the size of the construction program was to use the same subroutine 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 from the $NORTH$ of the memory — these are the flushed parts.

The complete construction program is too long to list here but can be found on the CD attached to this thesis. Two portions of the construction program are listed in Appendix B. The first listing is for the construction of a single memory module and shows how the use of subroutines allows a large structure to be constructed using a small number

of instructions. The second listing is for the construction of a mechanism for converting from the *pulse* signal representation to the *4-pulse* signal representation and shows how a structure with more complex connectivity requirements can be constructed.

Construction generally proceeds from $BACK$ to $FRONT$ in layers, and from $EAST$ to $WEST$ within each layer. The $BACK$ to $FRONT$ direction is dictated by the fact that the construction arm is at the $BACK$ of the parent machine, and the $EAST$ to $WEST$ direction is dictated by the design of the construction head.

There is one point in the construction program where the time that the construction operation happens is important. The value output by the *memory address counter* must be aligned with the contents of the *memory*. This is achieved by making sure that an EAST the instruction that causes the *memory address counter* to begin counting is at an address 118 words from a 256-byte boundary. The particular instruction in question happens to be the final EAST instruction within a shared subroutine called 'East3:'. When writing the construction program it is easy to move subroutines around in memory, so this subroutine was moved so that the EAST instruction is at address 8,054.

The child machine is constructed in a state where it is about to begin collecting parts, as though it had just executed a GATHER instruction. The final piece of the child machine that is constructed is the *detect* mechanism. Activating the child machine is a matter of injecting a signal into the *detect* mechanism which causes the child to begin moving in a $WEST$ direction looking for parts.

Before the construction program was written, it was not known whether or not it would actually be possible to construct a replica given the relatively small working area of the construction arm compared to the size of the machine. The fact that some structures have connectivity patterns that are difficult to construct also introduced uncertainty.

It turned out that it was possible to construct a replica machine which is functionally identical to the parent machine with the caveat that some subsystems were redesigned so as to make them easier to construct. When 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 function, 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.

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 — particularly for the differences in part connectivity. It would also disguise the fact that the CBlocks3D environment and the architecture of the machine impose restrictions that lead to these design compromises. 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.

5.13 Validating the design

The correctness of the design of the machine and of the construction program was 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.

As mentioned in section 5.12 the child machine is not identical to the parent machine so a third step was required to complete the validation process. 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 contained on the CD attached to this thesis.

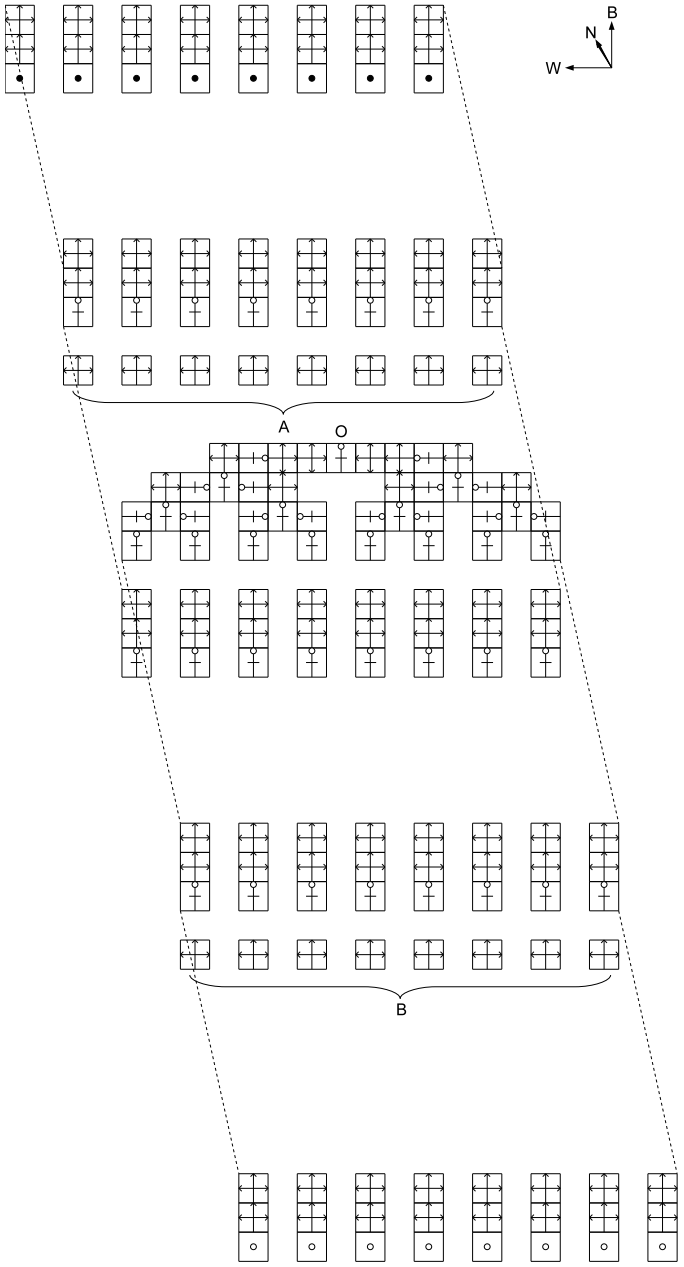


Figure 5.35. An 8-bit *comparator*.

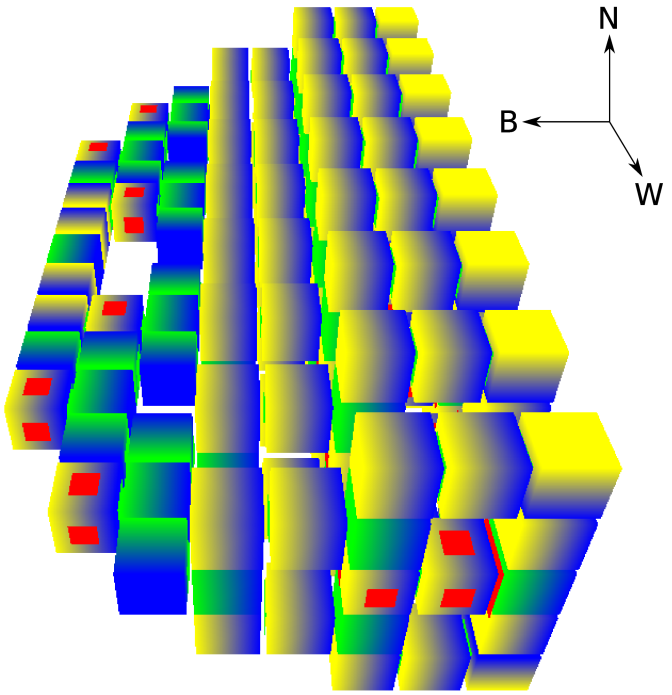


Figure 5.36. A graphical representation of the *comparator*.

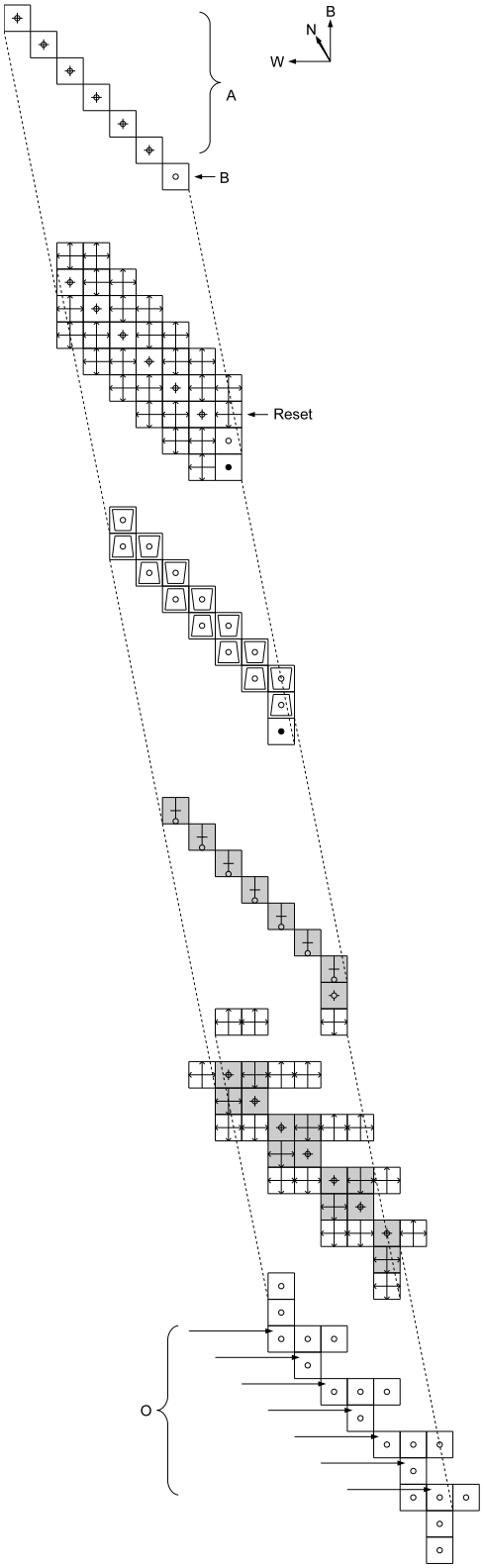


Figure 5.37. A *call register*.

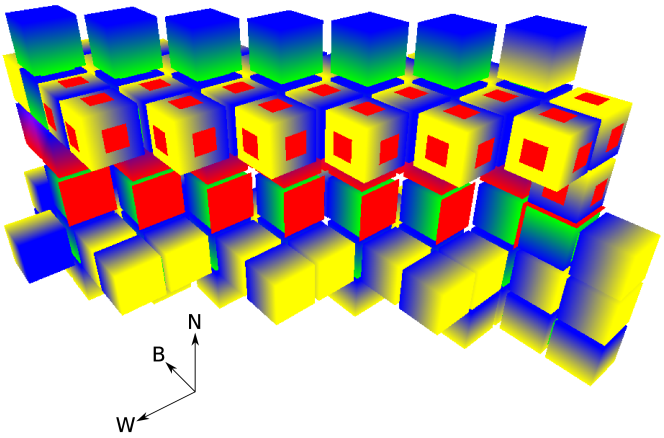


Figure 5.38. A graphical representation of a *call register*.

Chapter 6

Computing in Kinematic Environments

A kinematic environment has been defined as a system in which moveable parts interact with each other in some way and can be connected together to make larger structures. One of the considerations that led to the CBlocks3D system described in chapter 4 was that of attempting to reduce as far as possible the set of part types used by a self-replicating programmable constructor (SPRC) in a kinematic environment. In chapter 5 the information processing systems of an SRPC are made largely from *wire* and *nor* parts, but occasionally kinematic features of the environment are used to implement information processing mechanisms. For example the *address decoder* and the *call stack* both depend upon interacting moveable structures for their operation.

If it were possible to harness the *kinematic* behaviour of kinematic environments to implement all information processing mechanisms then this could potentially lead to a reduction in the number of part types needed to implement a machine like the one of chapter 5 by removing the need to have dedicated information processing parts such as the *wire* and *nor* parts.

It is also interesting from a physical perspective to consider whether an entirely mechanical SRPC in or near region G of Figure 2.18 on page 36 can be designed. It might be easier to do this than to design an electronic or electromechanical SRPC because electronic and electromechanical components tend to require high-precision processes to manufacture. If the manufacture of these components were beyond the scope of the SRPC itself, they would have to be provided as elementary parts and the resulting design would move away from region G along the *part complexity* axis.

6.1 Mechanical computing machines

The earliest computers were either wholly mechanical or contained a significant number of mechanical parts. Automated mechanical calculating aids have been around since the early 17th century [33]. In the mid 19th century Babbage's designs for his analytical engine [5] used columns of cogged wheels for storing decimal numbers. Operations on columns of wheels were carried out mechanically and the result of an operation automatically stored back into another column. The flow of a program in Babbage's machine could be made dependent on the result of a calculation, giving the engine the same computational power as a modern computer.

In the 1930s and 1940s Konrad Zuse designed and built several mechanical and electromechnical computers [53]. Zuse used a binary system for representing and storing numbers, which made his designs simpler and easier to build than Babbage's. His machines were also less ambitious than Babbage's, having a smaller memory and a less expressive instruction set. Table 6.1 compares the Z1 with the SRPC control unit described in section 5.11. This comparison indicates that the two systems have a similar complexity and that therefore it is reasonable to speculate that a machine no more complex than the Z1 could be used as the control unit for a mechanical SRPC.

Item	Z1	SRPC Control Unit
Instruction word size	8 bits	6 bits
Types of instruction	8	6
Program size	Unlimited tape length	8192 words
Memory size	64 words of 22 bits	8 words of 13 bits (call stack)
Component parts	Approx. 30,000	4,786 (not including memory)

Table 6.1. Comparison of Zuse's Z1 with the control unit of section 5.11.

Little theoretical work was carried out on mechanical computing before Zuse. It is a matter of historical conjecture that the advent of vacuum tubes and then transistors so soon after the potential of computers was recognised and the theoretical underpinnings of computing were established meant that mechanical computing machines never established themselves as they might have done had Babbage been successful in completing his analytical engine.

Recently there has been a resurgence of interest in the embodiment of computing systems in mechanical and other non-electronic substrates [30, 1].

In the interest of trying to reduce the number of parts that a system must support, investigations in this chapter adopt the principle of attempting to implement computing

schemes with the minimal number of types of part and the simplest types of interaction.

The billiard ball computing system of Fredkin and Toffoli [20] and similar schemes are a good example of this principle. However, this scheme was devised in order to investigate the theory of reversible computing and not as a practical system. There are some practical difficulties in implementing the billiard ball scheme. The first is that of positioning the billiard balls accurately and somehow making sure that any slight imperfections in collisions don't eventually end up causing the balls to be in the wrong position. The second is that once a configuration of balls has carried out a computation it can no longer be used. The system is a one-shot computing device.

The two results presented in this chapter, which have been published in [62] and [63], show how simple kinematic interactions can be used to implement reusable computing devices. Although both models are based in discrete space kinematic environments, the scheme in section 6.2 has a plausible physical implementation.

This thesis does not go as far as showing how these computing schemes can be used as the basis for information processing in an SRPC.

Software for simulating the systems described in this chapter can be found on the CD attached to this thesis.

6.2 Logic circuits in a system of repelling particles

The set of cellular automaton rules given in figure 6.1 specifies a simple behaviour: neighbouring tiles repel one another, but some tiles can be fixed in place.

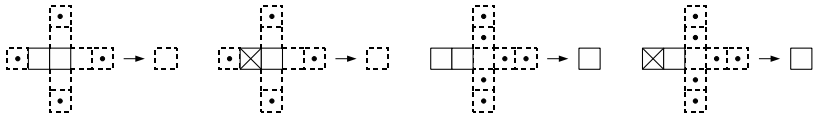


Figure 6.1. Tile behaviour specified as a set of cellular automaton rules

These rules specify a cellular automaton with an eight cell cross-shaped neighbourhood and three states per cell. A cell can be empty, or it can contain a fixed tile, or it can contain a moveable tile. In figure 6.1 empty cells are denoted by a square with a dashed boundary, fixed tiles are denoted by a square with a cross in, moveable tiles are denoted by a square with a solid boundary. Cells with a dot in the centre can be in any of the three states. The rules are symmetrical, so if a configuration of tiles rotated through any multiple of 90 degrees matches a rule, the central cell changes to the state to the right of the arrow on the next time step. If a configuration matches none of these rules, the central cell remains in the same state.

These rules are sufficient for all of the mechanisms described in this section.

6.2.1 Some basic mechanisms

Figures 6.2 to 6.10 show nine basic mechanisms from which more complex mechanisms can be put together.

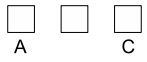


Figure 6.2. Wire

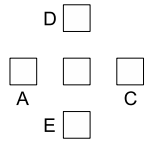


Figure 6.3. Cross

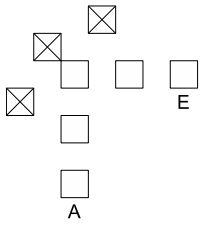


Figure 6.4. Corner (Type 1)

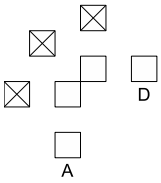


Figure 6.5. Corner (Type 2)

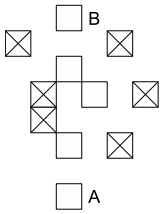


Figure 6.6. Changer

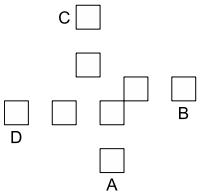


Figure 6.7. Fan-out

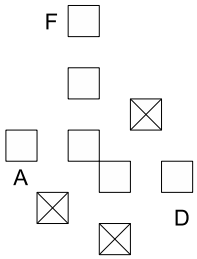


Figure 6.8. Combine

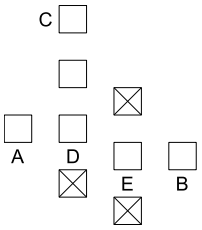


Figure 6.9. Both

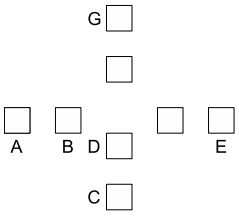


Figure 6.10. Hold

The notation used to describe the behaviour of each of these mechanisms will be introduced as it is used. The letters n,e,s and w that appear in square brackets in equations denote the directions north (up the page), south (down the page), east (to the right of the page) and west (to the left of the page).

6.2.1.1 Wire

For the Wire in figure 6.2 we can write:

$$A[e] \xrightarrow{2} C[e] \quad \begin{matrix} 1 \nearrow A[w] \end{matrix} \qquad (6.1)$$

$$C[w] \xrightarrow{2} A[w] \quad \begin{matrix} 1 \nearrow C[e] \end{matrix} \qquad (6.2)$$

$$A[e],C[w] \xrightarrow{1} C[e] \quad \begin{matrix} 1 \nearrow A[w] \end{matrix} \qquad (6.3)$$

Equation 6.1 states that the result of displacing A eastward by one unit at time t_A is that A will be displaced westward at $t_A + 1$, and C will be displaced eastward at $t_A + 2$. (The centre tile in the Wire will also move, but will return to its original position). A Wire can thus be thought of as a path along which a signal can propagate, where the signal consists of a displacement of a tile away from its normal position in the path. Wires of any length can be made.

Note that a Wire also works in reverse, as described by equation 6.2

Equation 6.3 describes what happens if both ends of a Wire are moved at once. This 'cancelling out' behaviour is used extensively by the logic gate described in section 6.2.3.

6.2.1.2 Cross

The Cross in figure 6.3 can be thought of as two wires crossing one another.

Note that the Cross mechanism misbehaves when $(A[e] \vee C[w]) \wedge (D[s] \vee E[n])$ at any time, so any circuit using the Cross mechanism must avoid this.

6.2.1.3 Corner (Type 1)

For the Type 1 Corner in figure 6.4 we can write:

$$\begin{array}{l} \overset{1}{\nearrow} A[s] \\ A[n] \xrightarrow{6} E[e] \end{array} \qquad (6.4) \qquad\qquad \begin{array}{l} \overset{1}{\nearrow} E[e] \\ E[w] \xrightarrow{6} A[s] \end{array} \qquad (6.5)$$

6.2.1.4 Corner (Type 2)

For the Type 2 Corner in figure 6.5 we can write:

$$\begin{array}{l} \overset{1}{\nearrow} A[s] \\ A[n] \xrightarrow{3} D[e] \end{array} \qquad (6.6) \qquad\qquad \begin{array}{l} \overset{1}{\nearrow} D[e] \\ D[w] \xrightarrow{3} A[s] \end{array} \qquad (6.7)$$

Note that both types of Corner can propagate signals in either direction.

6.2.1.5 Changer

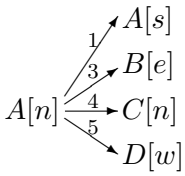
For the Changer in figure 6.6 we can write:

$$\begin{array}{l} \overset{1}{\nearrow} A[s] \\ A[n] \xrightarrow{7} B[n] \end{array} \qquad (6.8)$$

The Changer is so-called because it alters the spacing of the tiles in a signal path. This is often necessary when joining mechanisms together.

6.2.1.6 Fanout

For the Fanout mechanism in figure 6.7 we can write:

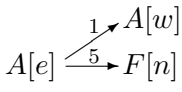


(6.9)

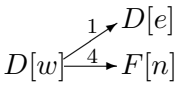
If we need fewer than 3 outputs, any of the output paths in the Fanout mechanism can be replaced with a single fixed tile.

6.2.1.7 Combine

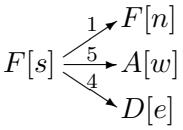
For the Combine mechanism in figure 6.8 we can write:



(6.10)



(6.11)

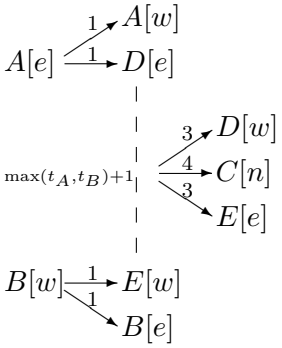


(6.12)

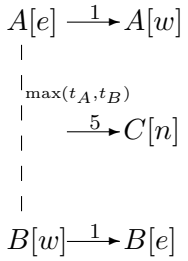
Equations 6.10 and 6.11 show that an output emerges from F when a signal is applied to either A or D. Equation 6.12 describes the behaviour of the Combine mechanism when driven from F.

6.2.1.8 Both

The Both mechanism will produce an output only after both of its inputs have been stimulated:



(6.13)



(6.14)

In equation 6.13 the dashed line with arrows leading to subsequent events indicates that the occurrence of two events causes the subsequent events (with subsequent events starting at time $\max(t_A, t_B) + 1$).

Because tiles D and E return to their original positions during the operation of this mechanism, it can be described more concisely by equation 6.14.

6.2.1.9 Hold

The Hold mechanism in figure 6.10 consists of four paths meeting at a junction, and is used as follows. At time t_A tile A may or may not be displaced eastward. At time t_E tile E may or may not be displaced westward. So that at time $t_m = \max(t_A, t_E) + 1$ the Hold mechanism may be in one of the four possible states shown in figures 6.10 to 6.13. At time $t_C \geq t_m$ tile C is displaced northward, and the response of the mechanism depends upon which of the four states it is in.

Let us call the arrangements shown in figures 6.10, 6.11, 6.12 and 6.13 H_0, H_1, H_2 and H_3 respectively.

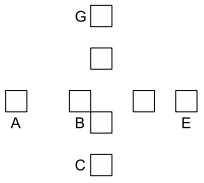


Figure 6.11. H_1

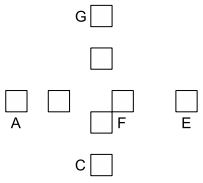


Figure 6.12. H_2

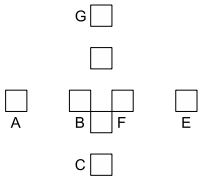


Figure 6.13. H_3

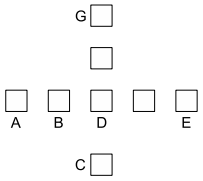


Figure 6.14. H_4

Equations 6.15 to 6.18 describe H_0, H_1, H_2 and H_3 respectively. What we have in effect is a mechanism that allows us to collide two signals (entering at A and E) together without having to worry about the relative timing of the two signals, because the collision only takes place when a signal is applied at C. When the Hold mechanism is used in a circuit, any signals emerging from A or E after the collision will propagate away from the Hold mechanism, leaving it in the state shown in figure 6.14, which we call H_4. We can return the mechanism to its original state H_0 by applying a signal at G.

$$\begin{array}{l} \overset{1}{\rightarrow} C[s] \\ C[n] \overset{1}{\rightarrow} D[n] \end{array} \qquad (6.15)$$

$$\begin{array}{l} \overset{1}{\rightarrow} C[s] \\ C[n] \overset{2}{\rightarrow} B[w] \\ \overset{4}{\rightarrow} E[e] \end{array} \qquad (6.16)$$

$$\begin{array}{l} \overset{1}{\rightarrow} C[s] \\ C[n] \overset{2}{\rightarrow} F[e] \\ \overset{4}{\rightarrow} A[w] \end{array} \qquad (6.17)$$

$$\begin{array}{l} \overset{1}{\rightarrow} C[s] \\ C[n] \overset{2}{\rightarrow} B[w] \\ \overset{2}{\rightarrow} F[e] \end{array} \qquad (6.18)$$

$$\begin{array}{l} \overset{1}{\nearrow} G[n] \\ G[s] \xrightarrow{2} D[s] \end{array} \tag{6.19}$$

Equation 6.19 describes the behaviour of H_4 when a signal is applied at G.

The fact that a signal will only be output at E if a signal is input at A but not at E before time t_C is the basis of the logic gate described in section 6.2.3.

6.2.2 Circuits

Circuits can be made by connecting mechanisms together, and in the next section a dual-rail logic gate is made using the nine mechanisms described in the previous section. Before doing this, it is necessary to show how to describe the behaviour of two mechanisms joined to one another in such a way that a signal emerging from one will enter another.

Figure 6.15 shows a Wire and a Combine mechanism that have tile B in common. We saw earlier how to describe the behaviour of each of these mechanisms. Equation 6.20 describes the behaviour of the joined mechanisms.

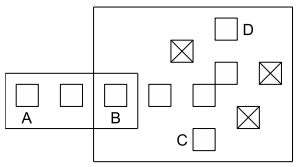


Figure 6.15. Two mechanisms joined via tile B

$$\begin{array}{l} \overset{1}{\nearrow} A[w] \quad \overset{1}{\nearrow} B[w] \\ A[e] \xrightarrow{2} B[e] \xrightarrow{5} C[s] \\ \qquad\qquad \underset{4}{\searrow} D[n] \end{array} \tag{6.20}$$

$$\begin{array}{l} \overset{1}{\nearrow} A[w] \\ A[e] \xrightarrow{7} C[s] \\ \underset{6}{\searrow} D[n] \end{array} \tag{6.21}$$

In words, equation 6.20 says that a displacement of A eastward by one unit at t_A causes A to be displaced westward at $t_A + 1$ and B to be displaced eastward at $t_A + 2$. The displacement of B subsequently causes B to be displaced westward at $t_A + 2 + 1$, C to be displaced southward at $t_A + 2 + 5$ and D to be displaced northward at $t_A + 2 + 4$.

Because there is no net movement of B, we can shorten equation 6.20 to equation 6.21.

6.2.3 A dual-rail logic gate

In subsection 6.2.1.9 we saw how the Hold mechanism effects a logical operation. In order to use this logical operation as the basis for a logic gate, additional circuitry is needed.

The A and E inputs to the Hold mechanism also serve as outputs after a signal has been applied at C. To enable us to extract a signal returning along a path which is also

used as an input we can use the mechanism shown in figure 6.16, called a 'uni-directional gate with tap'. An analysis of this mechanism is given in equations 6.22 and 6.23.

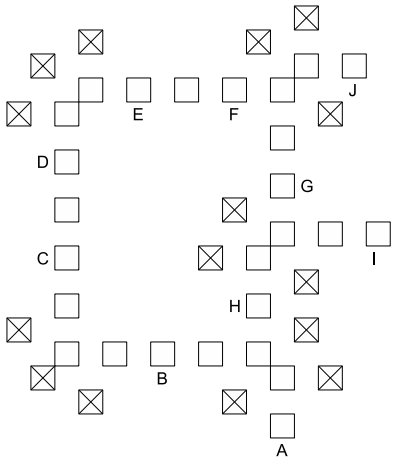


Figure 6.16. Uni-directional gate with tap

$$\begin{array}{ccccccccccccc} & \overset{1}{\nearrow} & A[s] & \overset{1}{\nearrow} & B[e] & \overset{1}{\nearrow} & C[s] & \overset{1}{\nearrow} & D[s] & \overset{1}{\nearrow} & E[w] & \overset{1}{\nearrow} & F[w] & \overset{1}{\nearrow} & G[n] \\ A[n] & \xrightarrow{4} & B[w] & \xrightarrow{6} & C[n] & \xrightarrow{2} & D[n] & \xrightarrow{3} & E[e] & \xrightarrow{2} & F[e] & \xrightarrow{5} & G[s] & \xrightarrow{5} & I[e] \end{array} \tag{6.22}$$

$$\begin{array}{ccccccccccc} & \overset{1}{\nearrow} & I[e] & \overset{1}{\nearrow} & G[s] & \overset{1}{\nearrow} & F[e] & \overset{1}{\nearrow} & E[e] & & & & \\ I[w] & \xrightarrow{5} & G[n] & \xrightarrow{5} & F[w] & \xrightarrow{2} & E[w] & \xrightarrow{3} & D[s] & & & & \\ & & & \searrow^{4} & J[e] & & & & & & & & \\ & & & & & & & & & \overset{1}{\nearrow} & D[n] & & \\ & \overset{4}{\searrow} & & & & & & & t_I+15 & \overset{1}{\nearrow} & C[s] & & \\ & & & & & & & & & & & & \\ & & & \overset{1}{\nearrow} & H[n] & \overset{1}{\nearrow} & B[e] & & & & & & \\ H[s] & \xrightarrow{5} & B[w] & & \xrightarrow{6} & & C[n] & & & & & & \end{array} \tag{6.23}$$

In equation 6.23 (as in equation 6.13) the dashed line with arrows leading to subsequent events indicates that the occurrence of two events (at time $t_I + 15$ in this case) causes the subsequent behaviour (in this case, the behaviour of a wire when both ends are displaced at once).

By observing which tiles return to their original positions during the course of the operation of this mechanism, we can shorten these equations to:

$$\begin{array}{ccccc} & \overset{1}{\nearrow} & A[s] & & \\ A[n] & \xrightarrow{27} & I[e] & & \end{array} \tag{6.24}$$

$$\begin{array}{ccccc} & \overset{1}{\nearrow} & I[e] & & \\ I[w] & \xrightarrow{9} & J[e] & & \end{array} \tag{6.25}$$

Thus, by applying a signal to A we can inject a signal onto a path connected to I. If a signal enters the mechanism at I, it will emerge at J.

Note that the number of tiles used in this mechanism could be reduced by shortening some of the paths. However, if we were to do this then we would not be able to analyse the mechanism in terms of the nine basic mechanisms described previously. To simplify the analysis and to make it clear which of the basic mechanisms we are using, we will use longer paths than are necessary.

By omitting the J output from the mechanism shown in figure 6.16, we can make a uni-directional gate described by equations 6.26 and 6.27.

$$A[n] \xrightarrow{1} A[s] \qquad A[n] \xrightarrow{27} I[e] \qquad (6.26)$$

$$I[w] \xrightarrow{1} I[e] \qquad (6.27)$$

Thus, the uni-directional gate will permit a signal to travel from A to I, but not in the reverse direction.

Recall that in dual-rail or 1-of-2 logic, every logical value X is represented by a pair of binary values $X_1 = X$ and $X_0 = \bar{X}$. We can implement a dual-rail logic scheme by using two signal paths corresponding to X_0 and X_1 in such a way that the passage of a signal along one path represents logic 0, and the passage of a signal along the other path represents logic 1. For a dual-rail logic gate with two logical inputs A and B (and therefore four signal path inputs A_0, A_1, B_0 and B_1), it is possible to detect when both logical inputs have been received using the mechanism shown in figure 6.17.

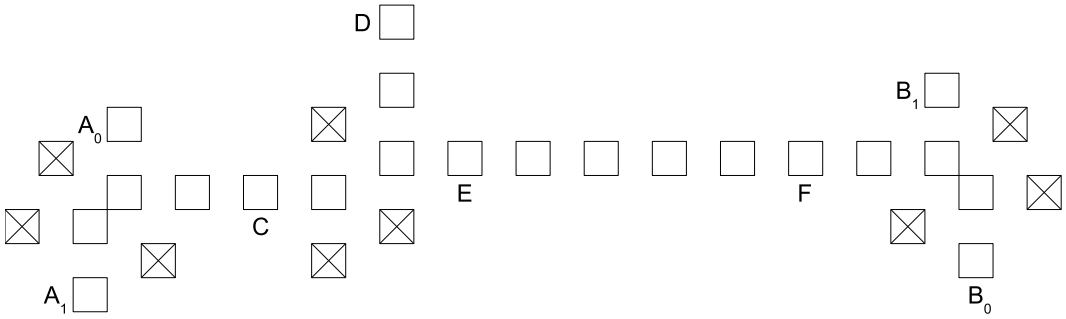
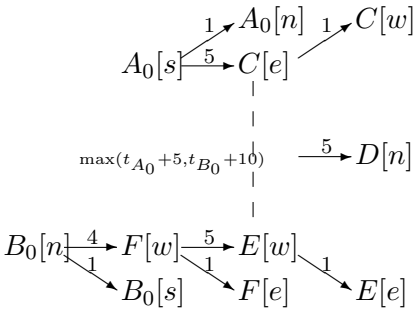


Figure 6.17. Dual-rail input detection mechanism

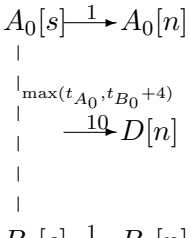
There are 4 possible cases to analyse for this mechanism, as shown in table 6.2. A complete analysis for the first case is given in equation 6.28, which can be shortened to equation 6.29. Complete analyses for the other three cases are omitted for the sake of brevity since they follow a similar pattern to equation 6.28.

A	B	A_0	A_1	B_0	B_1	Equation
False	False	1	0	1	0	6.29
False	True	1	0	0	1	6.30
True	False	0	1	1	0	6.31
True	True	0	1	0	1	6.32

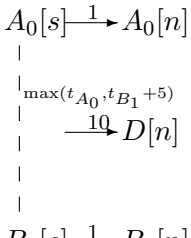
Table 6.2. Input cases for the Dual-Rail Input Detect mechanism



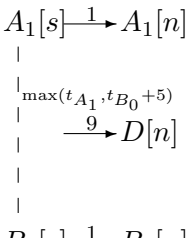
(6.28)



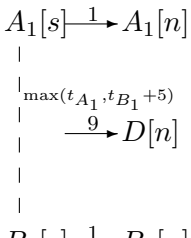
(6.29)



(6.30)



(6.31)



(6.32)

Equations 6.29 to 6.32 show that a signal will emerge at D for every possible combination of logical values that A and B can take.

When describing the Hold mechanism shown in figure 6.10 we noted that it will be used by firstly applying signals at A or E (or both, or neither), then applying a signal at C, and afterwards applying a signal at G to reset the mechanism. The augmented hold mechanism shown in figure 6.18 allows us to dispense with having to apply a reset signal by deriving a reset signal from the signal at C.

The net behaviour of the augmented hold mechanism is given in equations 6.33 to 6.36 for the four possible cases that result if signals are applied at A or B or both or neither.

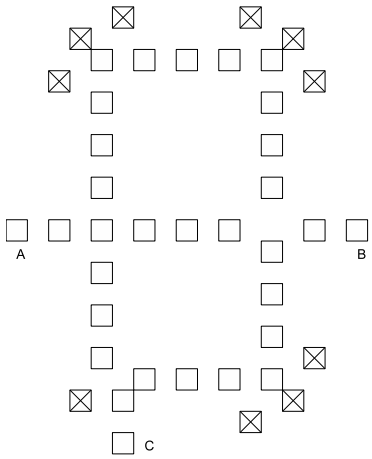


Figure 6.18. Augmented hold mechanism

$$\begin{array}{l} A[e] \xrightarrow{1} A[w] \\ \vdots \\ \vdots \xrightarrow{\max(t_A, t_C+5)} \\ \xrightarrow{8} B[e] \\ \vdots \\ \vdots \\ C[n] \xrightarrow{1} C[s] \end{array} \qquad (6.33)$$

$$\begin{array}{l} B[w] \xrightarrow{1} B[e] \\ \vdots \\ \vdots \xrightarrow{\max(t_B, t_C+5)} \\ \xrightarrow{8} A[w] \\ \vdots \\ \vdots \\ C[n] \xrightarrow{1} C[s] \end{array} \qquad (6.34)$$

$$\begin{array}{l} A[w] \xrightarrow{1} A[e] \\ B[w] \xrightarrow{1} B[e] \\ C[n] \xrightarrow{1} C[s] \end{array} \qquad (6.35)$$

$$C[n] \xrightarrow{1} C[s] \qquad (6.36)$$

Equation 6.35 holds so long as $t_C \geq \max(t_A - 5, t_B - 9)$.

Using a uni-directional gate with tap, a uni-directional gate, an augmented hold mechanism, a dual-rail input detection mechanism, two Combine mechanisms, a Changer and some connecting wires we can make the mechanism shown in figure 6.19. This mechanism feeds signals derived from A_0 and B_1 into an augmented hold mechanism, where a collision will take place once the dual-rail input detect mechanism indicates that all required inputs have been received. After the collision a signal will emerge at O if a signal was applied at A_0 but not at B_1. A signal derived from the output of the dual-rail input detect mechanism will emerge at P regardless of the logical values of A and B.

Equation 6.37 shows an analysis of the case for $A_0[e]$ at time t_{A_0} and $B_0[n]$ at time t_{B_0}.

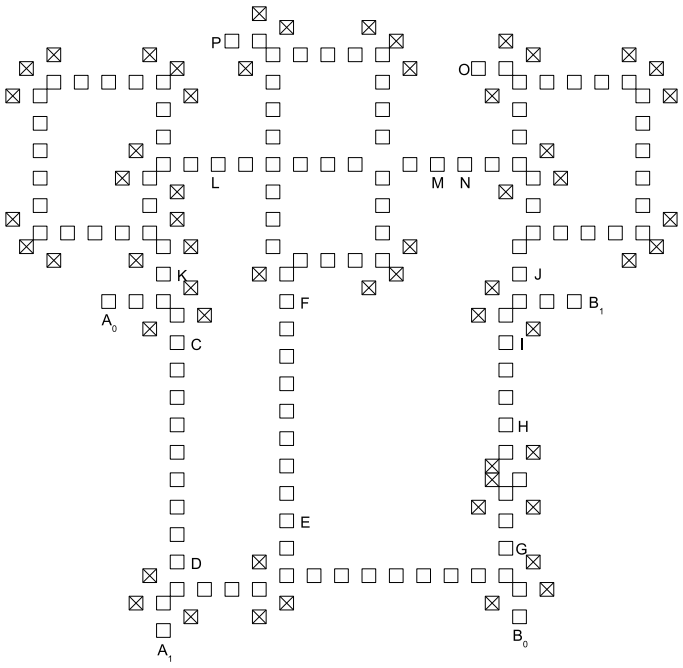
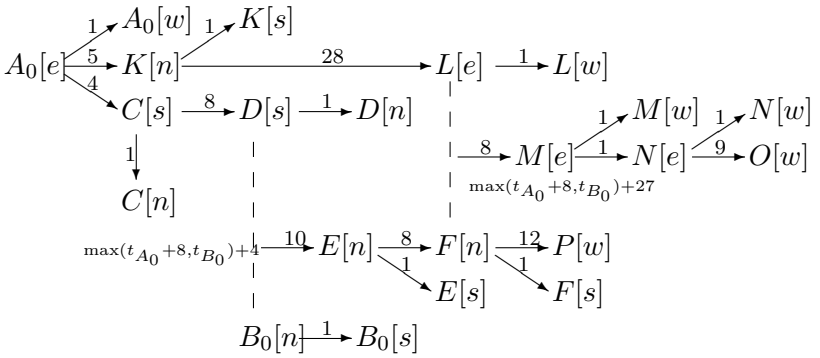
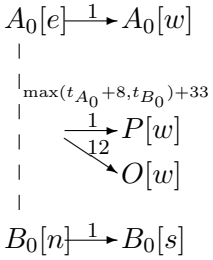


Figure 6.19. Logic gate - Stage 1

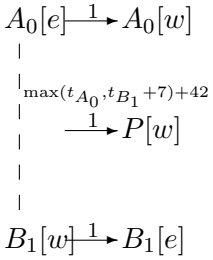


(6.37)

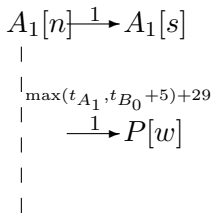
This can be simplified to equation 6.38. Equations for the other three cases are shown in 6.39 to 6.41. (Complete analyses for these are not given since they follow a similar pattern to 6.37).



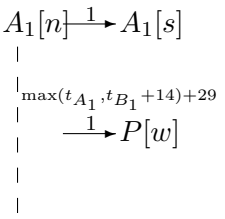
(6.38)



(6.39)



(6.40)



(6.41)

Thus in all four cases, a signal will emerge at P.

In the case shown in equation 6.38 (where signals are applied at A_0 and B_0), a signal will emerge at P at time $t_P = \max(t_{A_0} + 8, t_{B_0}) + 34$ and another signal will emerge at O at time $t_O = \max(t_{A_0} + 8, t_{B_0}) + 45 = t_P + 11$.

The mechanism shown in figure 6.20 can be used to derive a signal from O to use as the Q_1 output of a dual-rail logic gate, and to derive a signal from P and O to use as the Q_0 output of a dual-rail logic gate.

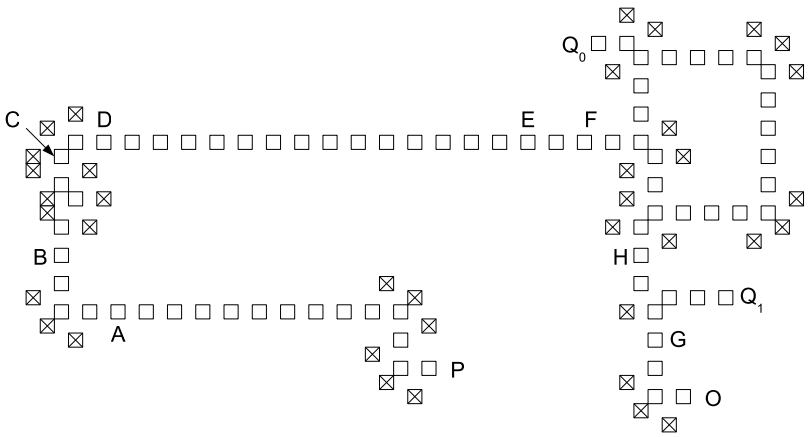
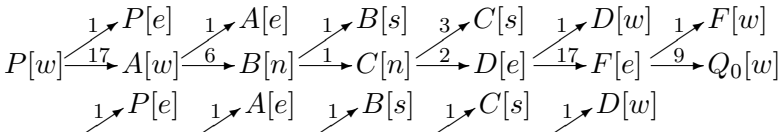
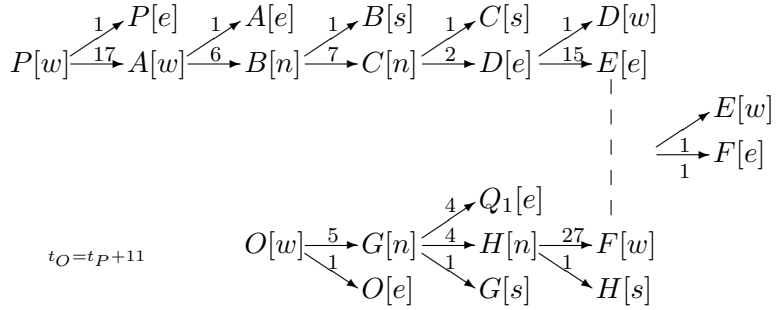


Figure 6.20. Logic gate - Stage 2

An analysis of this mechanism is given in equations 6.42 and 6.43.



(6.42)



(6.43)

These equations can be simplified to 6.44 and 6.45 and used in conjunction with 6.38 to 6.41 to deduce the overall behaviour of the logic gate.

$$\begin{array}{l} \overset{1}{\nearrow} P[e] \\ P[w] \xrightarrow{54} Q_0[w] \end{array} \qquad (6.44)$$

$$\begin{array}{l} P[w] \xrightarrow{1} P[e] \\ \\ O[w] \xrightarrow{1} O[e] \end{array} \qquad \begin{array}{l} \\ \overset{9}{\nearrow} Q_1[e] \\ \\ \end{array} \qquad (6.45)$$

A	B	Inputs to gate	Outputs from gate	Q
F	F	A_0 at t_{A_0} , B_0 at t_{B_0}	Q_1 at $\max(t_{A_0}+8, t_{B_0})+92$	T
F	T	A_0 at t_{A_0} , B_1 at t_{B_1}	Q_0 at $\max(t_{A_0}, t_{B_1}+7)+97$	F
T	F	A_1 at t_{A_1} , B_0 at t_{B_0}	Q_0 at $\max(t_{A_1}, t_{B_0}+5)+84$	F
T	T	A_1 at t_{A_1} , B_1 at t_{B_1}	Q_0 at $\max(t_{A_1}, t_{B_1}+14)+84$	F

Table 6.3. Truth table for the logic gate

Thus, the overall behaviour of our dual-rail logic gate is described by table 6.3. From this, it can be seen that the logic gate is a dual-rail implementation of a boolean NOR gate.

6.2.4 Summary

The number of tiles in the logic gate could be reduced by shortening the paths between mechanisms and by reducing the size of some mechanisms which were deliberately kept larger than necessary in order to clarify their structure. Unused fixed tiles could also be removed at some corners and in some Combine mechanisms.

The part of the logic gate that performs the logic operation is the augmented hold mechanism. If we placed constraints on signal timing at the inputs to the gate, we could do without this mechanism and replace it with a wire, resulting in a simpler gate. However, if we were to do this and then attempt to connect several logic gates together to make a circuit we would have to introduce delays between one gate and another in order to meet timing constraints.

The following arrangement could be used to make a physical implementation of this system: There is a regular lattice of traps in which particles are held fixed in place. Traps are disengaged, particles are released and repel one another. When enough time has passed for any neighbouring particles to have moved to adjacent lattice points, the traps are reengaged to trap the particles again. The force-distance profile of the repulsive force between particles must be such that particles at neighbouring lattice points are strongly repelled, but particles two or more lattice points apart experience little mutual repulsion.

6.3 A kinematic Turing machine

This section shows how a Turing machine can be made using kinematic interactions in the CBlocks3D environment. The Turing machine presented here has some similarities to that described by Laing in [35]. The most notable similarity is the way in which state transitions are implemented. In a state diagram, state transitions can be represented as arrows that lead from one state to another. In [35] and in the system described here, state transitions are embodied as paths that run from one state to another that get followed by the Turing machine.

All mechanisms are made using just one component part similar to the *slide* part defined in Table 4.1 on page 68, but which is always active.

6.3.1 Notation and Petri net diagrams

Figures 6.21 and 6.22 show successive states S_n and S_{n+1} of a universe.

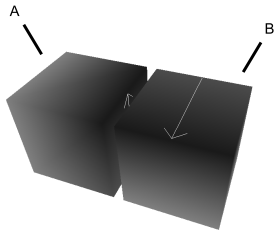


Figure 6.21. A simple mechanism in state S_n. Constructs A and B are labelled.

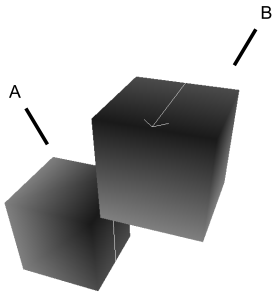


Figure 6.22. A simple mechanism one time unit later in state S_{n+1}.

Figure 6.23 is a state diagram describing the system. S_n and S_{n+1} are represented by circles. The transition between S_n and S_{n+1} is represented by an arrow, labelled with the interaction between constructs that causes the state to change. In this case, the interaction

is $A, B \rightarrow B[n]$. Meaning 'An interaction between constructs A and B causes B to move North'. The whole diagram means 'When the state of the system is S_n, an interaction between A and B causes B to move North, and the system ends up in state S_{n+1}'.

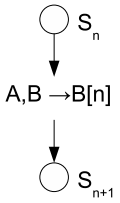


Figure 6.23. State diagram for a simple mechanism.

For systems containing several interacting mechanisms, a method for describing interacting state diagrams is required. Petri nets [50] are suitable for this purpose.

Consider the interacting mechanisms shown in figure 6.24. The behaviour of the mechanism labelled CYC can be described by the state diagram shown in figure 6.25. This mechanism is not influenced in any way by its neighbouring mechanism, FLIP. On the other hand, construct D in mechanism FLIP is influenced by construct B in mechanism CYC. A Petri net for the complete system is shown in figure 6.26. A Petri net consists of conditions which are represented by circles, and transitions which are represented here by a description of the action corresponding to the transition. Any number of conditions in a Petri net can be 'marked'; a marked condition is represented by a dot within a circle. Pre-conditions of a transition T are those conditions that have arrows leading to T. Arrows leading from T lead to post-conditions of T. A transition can occur whenever all pre-conditions of the transition are marked. When a transition takes place all of its pre-conditions become unmarked and all of its post-conditions become marked. The Petri nets used in this section are deterministic: whenever a transition can occur it does occur.

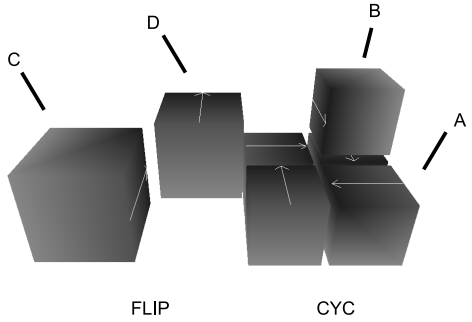


Figure 6.24. Two interacting mechanisms.

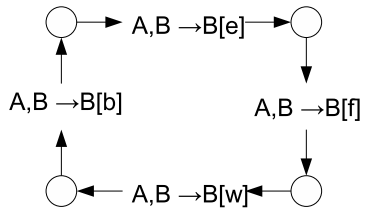


Figure 6.25. State diagram for CYC mechanism.

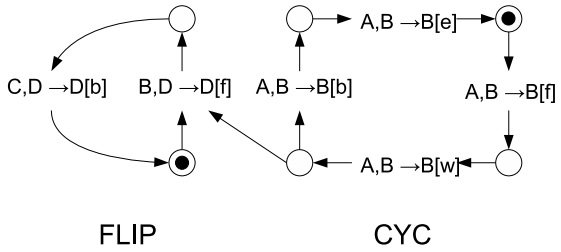


Figure 6.26. Petri net diagram for CYC and FLIP mechanisms.

6.3.2 Turing machines - definitions

The following formal description of a Turing machine is used.

Set of symbols $\Sigma = \{0, 1\}$
Set of states Q
Transition function $\delta : Q \times \Sigma \to Q \times \Sigma \times \{-1, 1\}$
Starting state q_0
Halting state q_h

The configuration C_n of a Turing machine at a particular step n in its evolution is defined as follows:

$C_n.q \in Q$ is the state of the machine at step n.
$C_n.a_1...a_t \in \Sigma^t$, where $t \in \mathbb{N}$, are the contents of the tape at step n.
$C_n.i$ is the position of the head on the tape at step n.

$C_n.a_i$ is used as an abbreviation for $C_n.a_{C_n.i}$.

The initial configuration of the machine is:

$C_0.q = q_0$
$C_0.i = 1$
$C_0.a_1...a_t$ are the initial contents of the tape.

The evolution of a Turing machine over time is given by equation 6.46. Any contents of the data tape not changed by equation 6.46 remain unchanged from step n to step $n+1$.

$$(C_{n+1}.q, C_{n+1}.a_{c_n.i}, C_{n+1}.i - C_n.i) = \delta(C_n.q, C_n.a_{c_n.i}) \tag{6.46}$$

$\delta(q,a).state$, $\delta(q,a).symbol$ and $\delta(q,a).direction$ are used when it is necessary to refer to individual elements from the tuple $\delta(q,a)$.

The Turing machine halts when $C_n.q = q_h$, and C_n is the final configuration of the machine.

6.3.3 A Turing machine in the CBlocks3D environment

6.3.3.1 Overview

Figure 6.27 gives an overview of the Turing machine presented in this section.

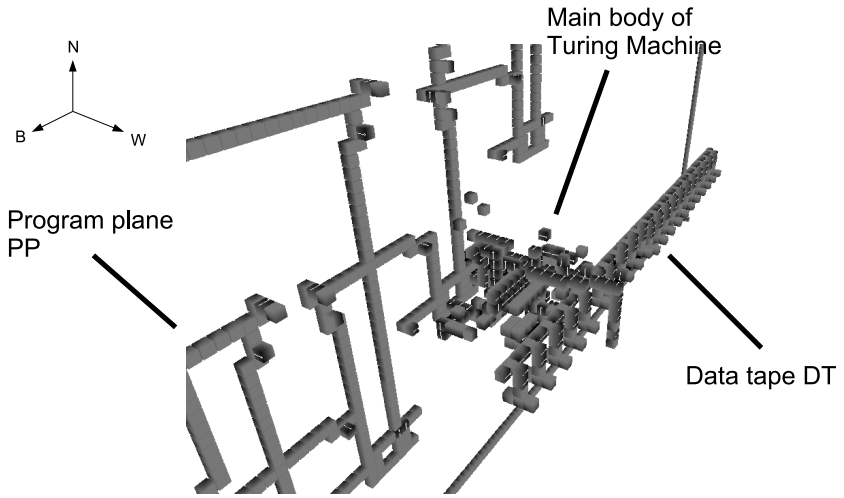


Figure 6.27. Turing machine - Overview

The Turing machine contains the following mechanisms, each of which is described in detail later on:

- The data tape mechanism DT: This implements a sequence of bits corresponding to the Turing machine's tape.
- The program plane construct PP: This implements the transition function δ. For every $q \in Q$ there is a corresponding position of the program plane PP. Let $F : Q \to F \times F$ be the function that maps Turing machine states to program plane positions. A state transition is carried out by moving PP from one position to another. Information encoded on PP tells the machine how to respond when a 0 bit or a 1 bit is encountered on the data tape DT. PP also contains parts that trigger the sequencer SQ via the trigger mechanism TR when a state transition has finished.
- The bit-reading mechanism BR: Interrogates the current position on the data tape DT and activates the selection mechanism SL if a 1 bit is encountered.
- The selection mechanism SL: This moves the program plane PP so as to select the second of two sets of information encoded for a particular state on PP.
- The condition-action mechanism CA: Interrogates the program plane PP and performs bit-writing and/or tape-moving actions according to information associated with the current state on PP.
- The conditional-tape-moving mechanism CTM: Moves the data tape DT six units backward if activated by the condition-action mechanism CA.
- The unconditional-tape-moving mechanism UTM: Moves the data tape DT three units forward when activated by the sequencer SQ.
- The sequencer mechanism SQ: Triggers various parts of the machine in turn in order to perform a cycle of operation. Triggers UTM, then BR, then CA, then PM, then waits to be reset by the program plane PP after a state transition has occurred.
- The plane-moving mechanism PM: Follows tracks on the program plane PP to effect a transition from one state to another by moving PP.
- The trigger mechanism TR: Activated by the program plane PP when a state transition has finished in order to reset the sequencer SQ.

Section 6.3.4 contains illustrations for all of these mechanisms, along with Petri net diagrams which concisely describe the operation of each mechanism.

6.3.3.2 Operation

Suppose that the Turing machine is in a state corresponding to the configuration C_n. Then the z coordinate of the data tape DT is related to $C_n.i$ by $DT.z = 3 \times C_n.i + A$ where A is a constant that depends on the absolute location of the Turing machine in space, and bit $C_n.a_p$ is represented by pin $DT.B_p$.

The main body of the Turing machine is made from mechanisms BR, SL, BW, CTM, PM, UTM, SQ and TR and always remains stationary (apart from movements of internal mechanisms). Only DT and PP move around.

The operation of the machine is as follows:

1. The machine is in a state corresponding to a configuration C_n.
2. SQ triggers BR, CA, PM and UTM in turn.
3. BR reads the current symbol from DT. If the symbol is 0, SL does not move PP, otherwise SL moves PP four units backward.
4. CA first examines PP to determine $\delta(C_n.q, C_n.a_i).symbol$, and then sets the current symbol on DT to this.
5. CA then examines PP to determine $\delta(C_n.q, C_n.a_i).direction$ and hence which direction to move DT in. It will either leave DT as it is, or move DT six units backward.
6. PM causes PP to move along a path from $F(C_n.q)$ to $F(\delta(C_n.q, C_n.a_i).state)$
7. When PP has moved to this new position, PP resets SQ.
8. SQ triggers UTM and UTM moves DT three units forward.
9. The machine is now in a state corresponding to configuration C_{n+1}. Operation continues at step 1.

So, by the end of a cycle of operation, the following has happened:

Depending on the current location of PP (corresponding to $C_n.q$) and the position of the pin $DT.B_{C_n.i}$ at the current location on DT (corresponding to $C_n.a_i$), a symbol corresponding to $\delta(C_n.q, C_n.a_i).symbol$ has been written to DT, DT has been moved 3 places forwards or backwards, corresponding to $\delta(C_n.q, C_n.a_i).direction$, and PP has been moved to a new position corresponding to $\delta(C_n.q, C_n.a_i).state$. This action corresponds to the abstract operation described by equation 6.46.

The machine can be made to halt by having a path on PP that leads nowhere. When the machine tries to move PP to a new state by following this path, the operation of the machine will cease.

6.3.4 Detailed description of mechanisms

6.3.4.1 Data Tape DT

Figure 6.28 shows part of the mechanism *DT* for representing a sequence of binary digits. Figure 6.29 shows the same mechanism viewed from the opposite direction. The mechanism consists of a long rod A situated against a series of pins B_x that can be individually raised or lowered with respect to the rod to represent binary 1 and 0 digits respectively. The mechanism is designed so that when the rod A is moved east or west, the pins get moved along with it so that the representation is not disturbed.

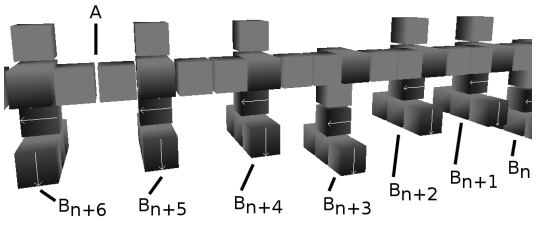


Figure 6.28. The DT mechanism for representing a binary string. (View 1)

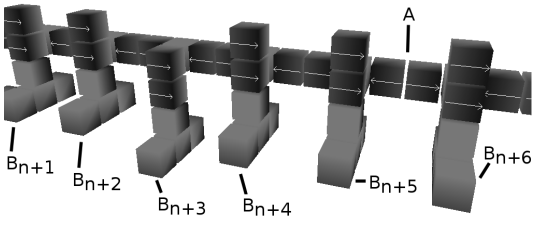


Figure 6.29. The DT mechanism for representing a binary string. (View 2)

Figures 6.30 and 6.31 shows Petri nets for the data tape DT. Note that DT has a very large number of possible states, equal to the number of possible z-coordinates of DT multiplied by the number of integers that all of the bits on DT can represent. Because of this, figures 6.30 and 6.31 use a pair of integers i, j to represent any one of a large set of possible states, where i corresponds to the number represented by all of the bits on DT (and so setting or resetting a bit corresponds to setting $i := i + 2^n$ or $i := i - 2^n$), and

where j corresponds to the z-coordinate of DT.

The large, dashed-outline circles in figures 6.30 and 6.31 represent states in other Petri nets. For example, in figure 6.30, the dashed circle labelled with UTM_1 represents the state labelled 1 in the Petri net for UTM (figure 6.43).

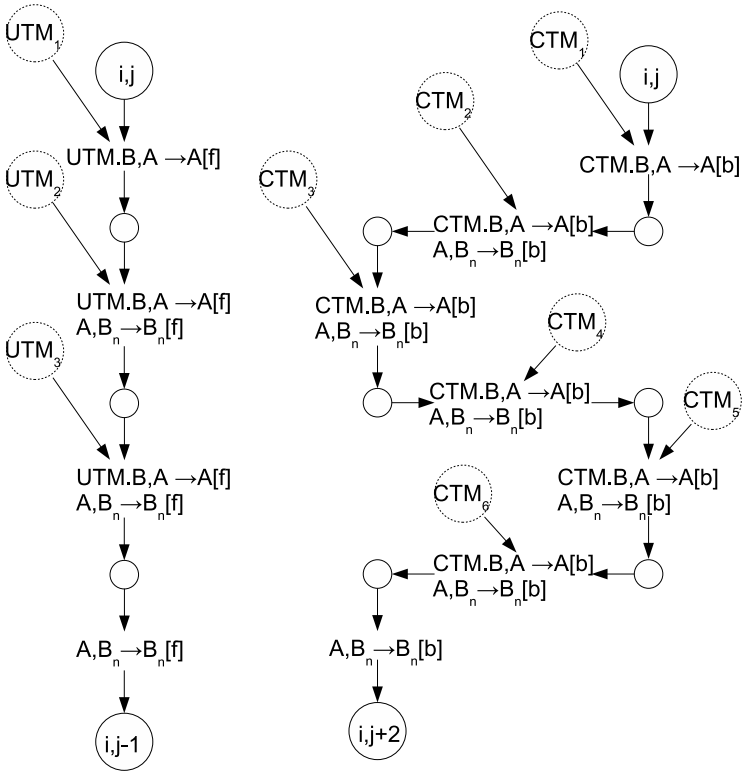


Figure 6.30. Petri net for DT. Moving back and forth.

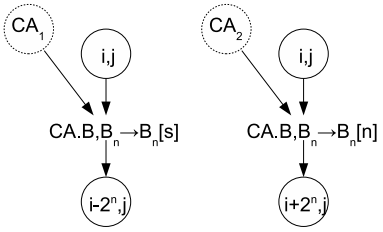


Figure 6.31. Petri net for DT. Resetting and setting a bit.

6.3.4.2 Bit Reading mechanism BR

Figure 6.32 shows a mechanism for reading a bit on the data tape. Figure 6.33 shows an exploded view of the same mechanism in which the different constructs that make up the mechanism can be distinguished.

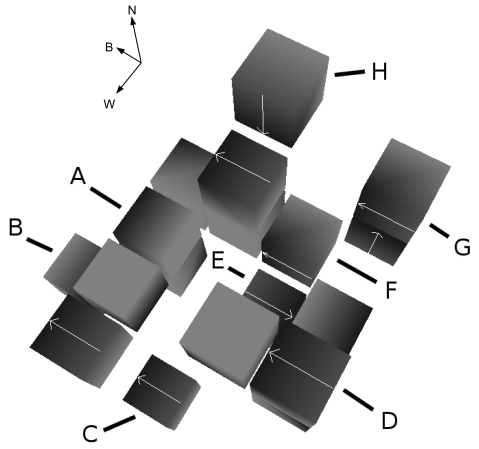


Figure 6.32. The BR mechanism for detecting the state of a single bit on a data tape.

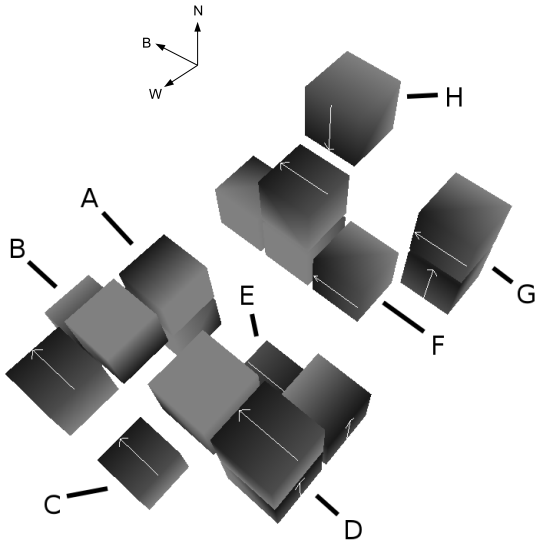


Figure 6.33. Exploded view of figure 6.32.

Figure 6.34 shows the Petri net for the Bit Reading mechanism BR, which explains its operation. Note that this Petri net uses an 'inhibit' arc (denoted by a line terminated with a circle). A transition that has an inhibit arc as an input cannot take place if the

state from which the inhibit arc emanates is marked.

In the Petri net for BR, the inhibit arc is used so that the network 'chooses' between the path which corresponds to a 1 bit being read from DT, and the path which corresponds to a 0 bit being read from DT. $DT.B_n$ refers to the pin of DT that is currently adjacent to BR and DT_x refers to a state of DT in which $DT.B_n$ is in a raised position representing the binary digit 1.

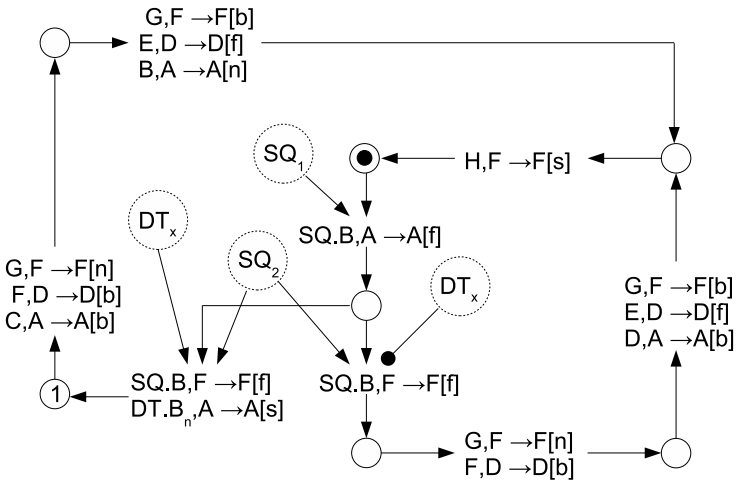


Figure 6.34. Petri net for BR.

6.3.4.3 Selection mechanism SL

Figure 6.35 shows the mechanism that selects the second of two positions for a given state on the program plane, if it is activated by BR. Consider the sentence: 'If the Turing machine is in state q and encounters a 0, then do ..., otherwise if it encounters a 1 then do ...'. The SL mechanism is the implementation of the word 'otherwise' in this sentence.

Figure 6.36 shows the Petri net for the SL mechanism.

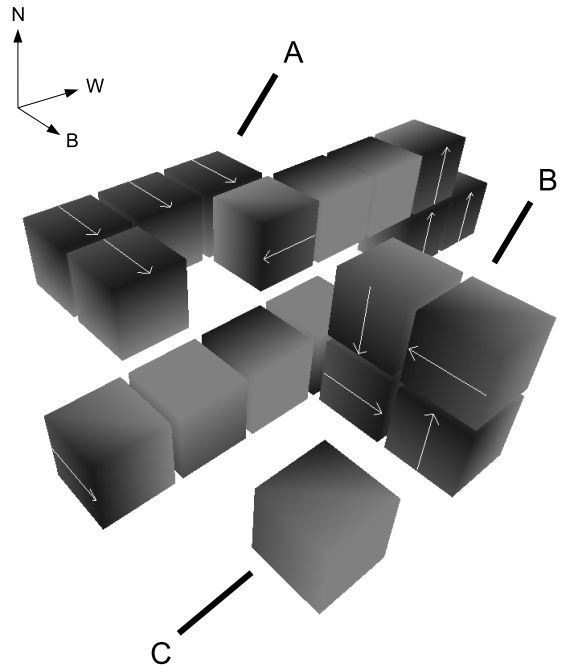


Figure 6.35. The SL mechanism for conditionally moving the program plane.

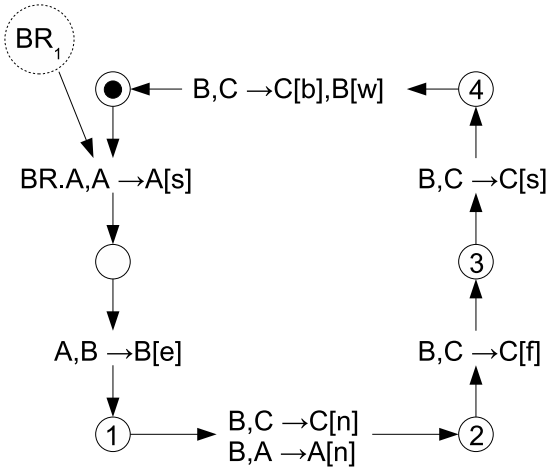


Figure 6.36. Petri net for SL.

6.3.4.4 Condition Action mechanism CA

After the Selection Mechanism SL has ensured that the correct part of the current state on the program plane PP is in the right place, the CA mechanism interrogates that place on PP to work out whether to write a 1 or a 0 onto the data tape DT, and whether to move DT backwards or forwards.

Figures 6.37 and 6.38 show the CA mechanism. Basically it consists of two rods, one rod (labelled B) for determining the state of the 'Write a 1' bit on the program plane. The other rod (labelled A) for determining the state of the 'Advance the tape' bit on the program plane. Each rod is pressed against the program plane, and the corresponding action is triggered.

Figure 6.39 shows the Petri net for the CA. The Petri net is complex because the mechanism consists of three main constructs - the two rods A and B, and a construct G that is responsible for moving rod B back to its initial position once PP has been interrogated. Some arcs on the Petri net that would show dependencies between the three loops in figure 6.39 have been omitted to avoid clutter, since they do not affect the behaviour of the net.

6.3.4.5 Conditional Tape Moving mechanism CTM

Figure 6.40 shows the CTM. This mechanism is triggered by the CA when the program plane PP indicates that the data tape DT is to be moved. The CTM moves DT 6 units

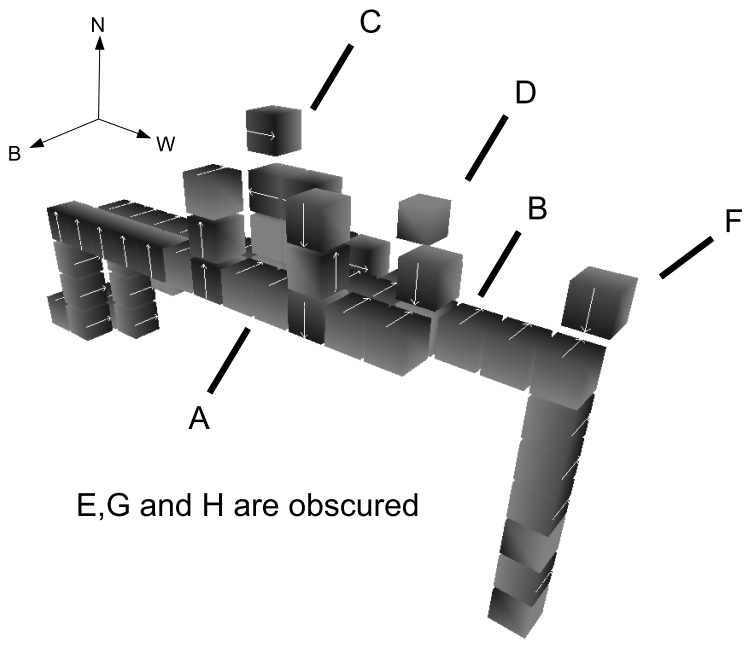


Figure 6.37. The CA mechanism that performs actions depending on information encoded on the program plane.

backwards, so that between them CTM and UTM (see the next subsection) move the DT either 3 units backwards or 3 units forwards.

Figure 6.41 shows the Petri net for the CTM.

6.3.4.6 Unconditional Tape Moving mechanism UTM

Figure 6.42 shows the UTM. This mechanism is triggered by the sequencer to move the DT 3 units forwards.

Figure 6.43 shows the Petri net for the UTM.

6.3.4.7 Sequencer SQ

Figures 6.44 and 6.45 show the Sequencer SQ which is responsible for activating several other mechanisms: BR, CA, PM and UTM.

It can be seen that SQ is essentially a track A along which a construct B moves. In several places B encounters other constructs (not shown in figures 6.44 and 6.45) and activates the mechanisms to which they belong. Figure 6.46 shows the Petri net for SQ.

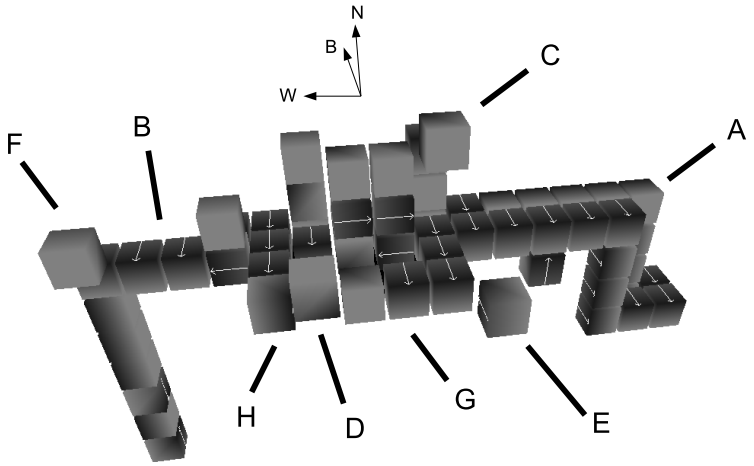


Figure 6.38. A different view of the CA mechanism

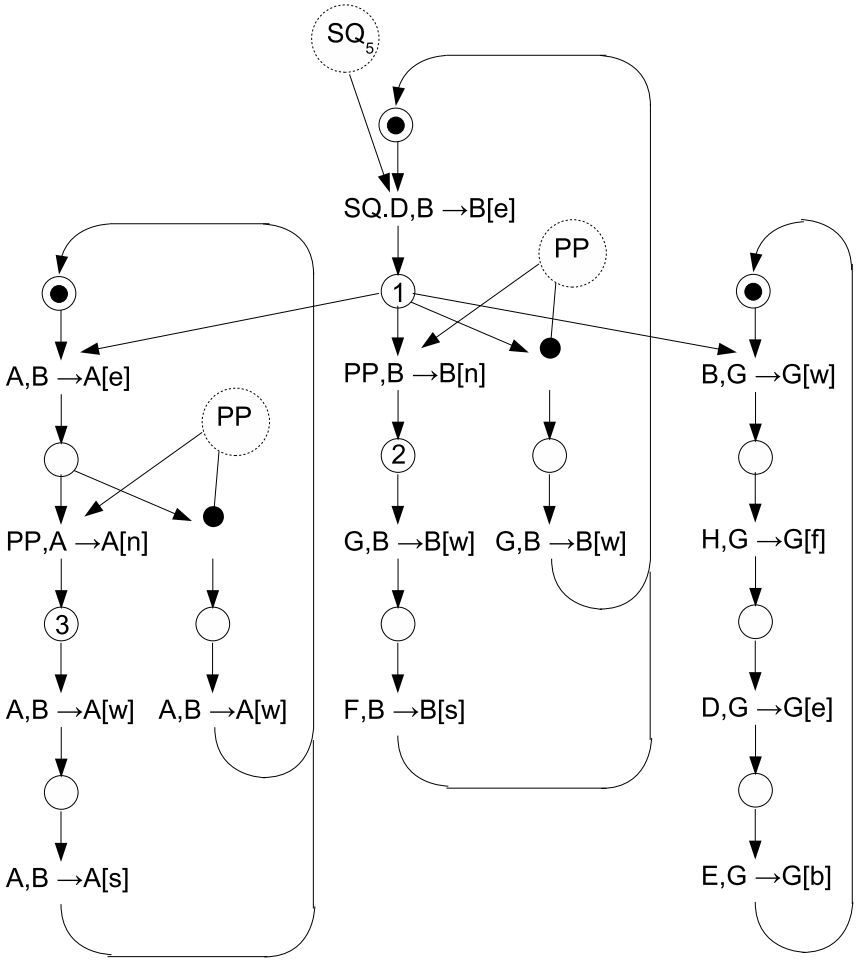


Figure 6.39. Petri net for CA.

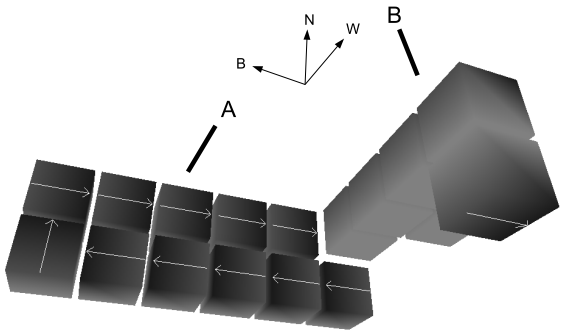


Figure 6.40. The CTM mechanism

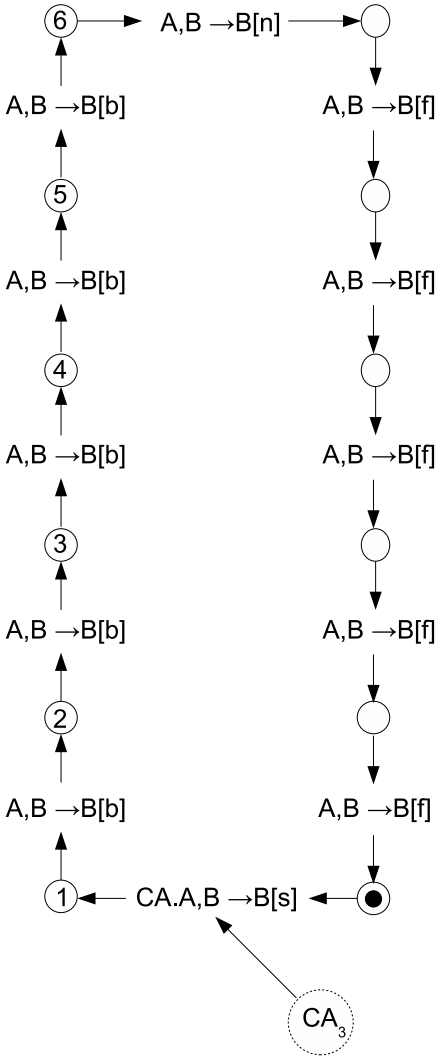


Figure 6.41. Petri net for CTM.

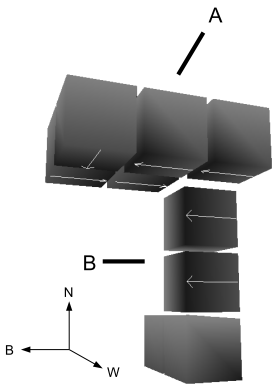


Figure 6.42. The UTM mechanism.

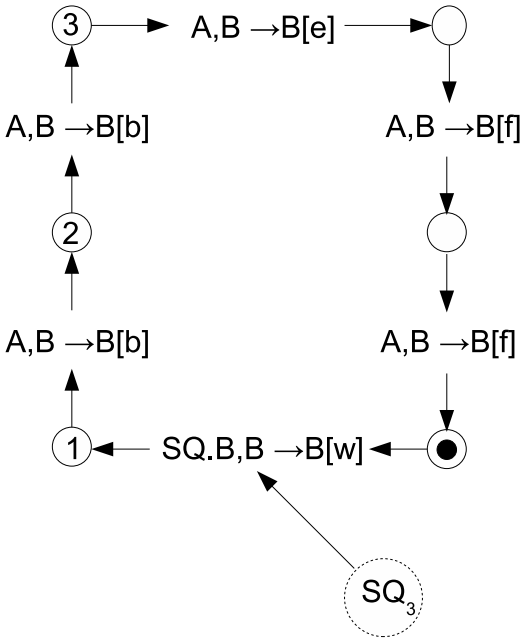


Figure 6.43. Petri net for UTM.

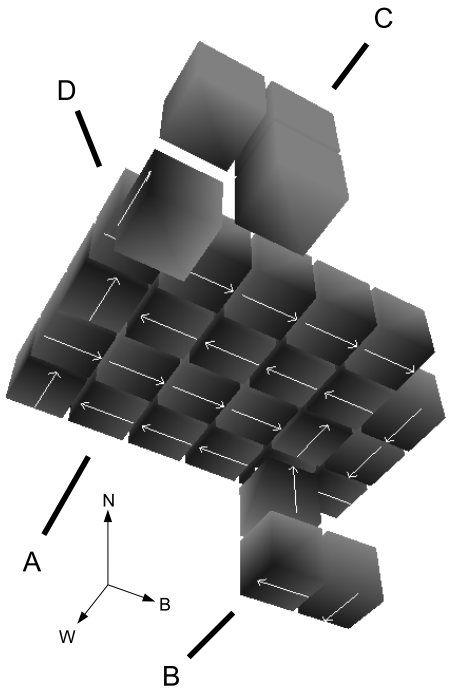


Figure 6.44. The SQ mechanism.

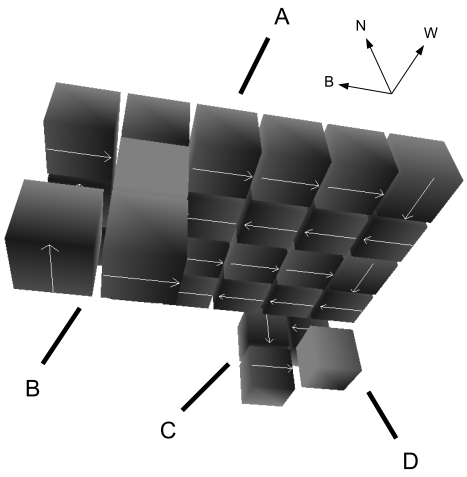


Figure 6.45. A different view of the SQ mechanism.

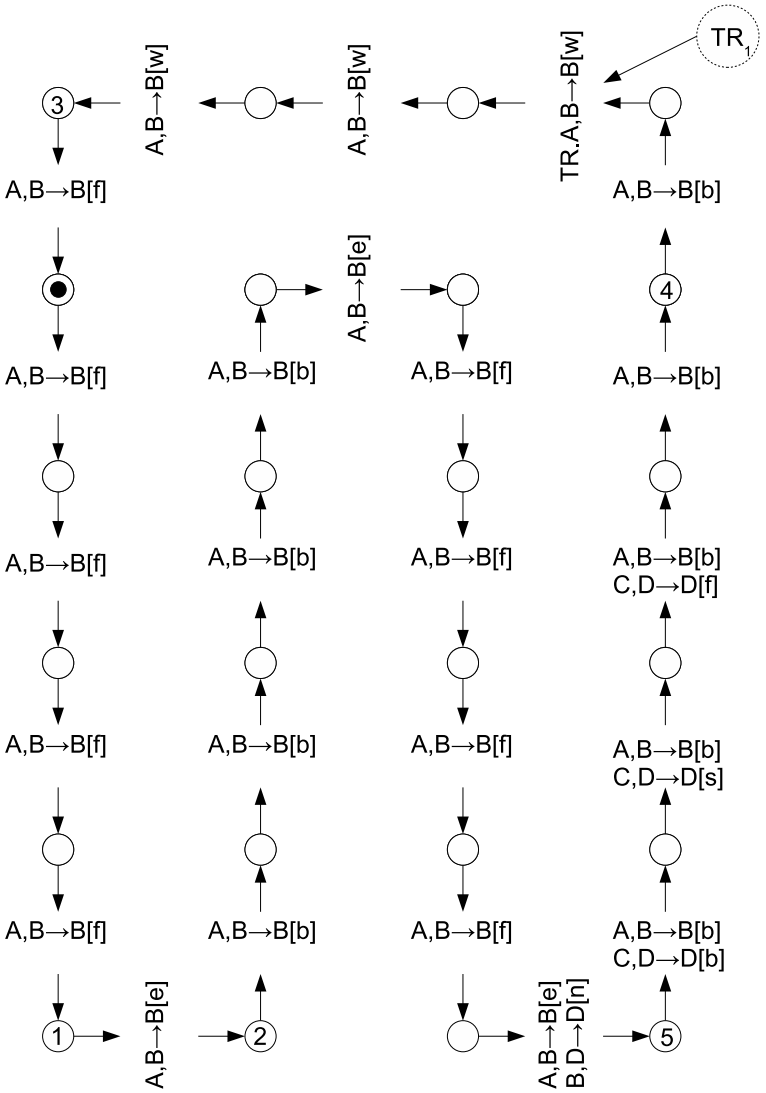


Figure 6.46. Petri net for SQ.

6.3.4.8 Plane Moving mechanism PM

Figures 6.47 and 6.48 show the plane moving mechanism PM. This mechanism gets activated by SQ after all actions related to reading and acting on information encoded at the current position of the program plane PP have finished. The mechanism tracks paths on the program plane PP that lead from the portion of PP that has just been examined to another state position on PP.

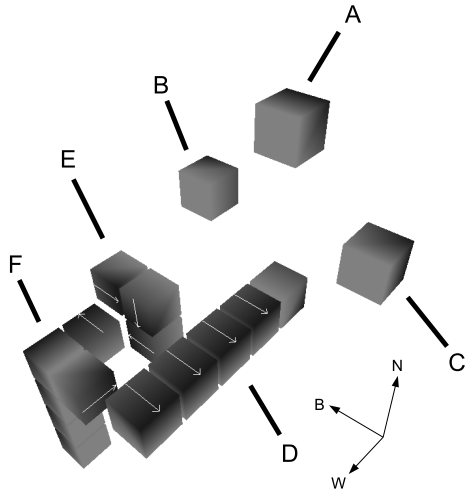


Figure 6.47. The PM mechanism.

Figure 6.49 shows the Petri net for PM. To avoid clutter in this diagram, not all PP states that have arcs leading to transitions are shown.

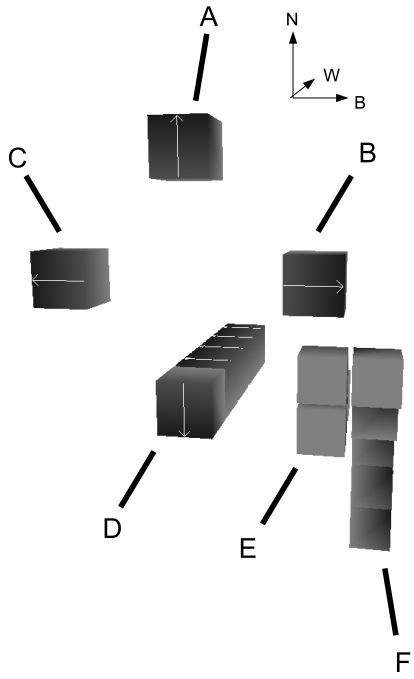


Figure 6.48. A different view of the PM mechanism.

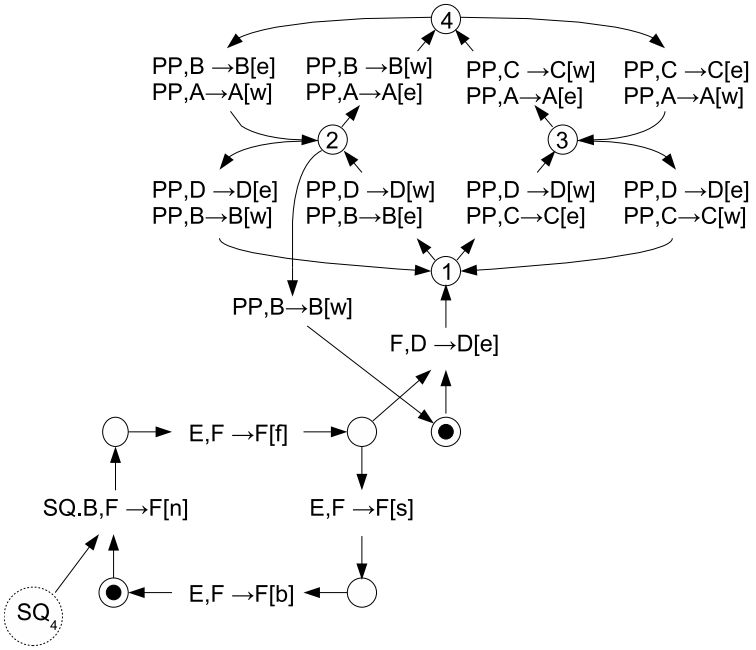


Figure 6.49. Petri net for PM.

6.3.4.9 Trigger mechanism TR

Figure 6.50 shows the trigger mechanism TR that starts the sequencer when the program plane PP reaches a state position.

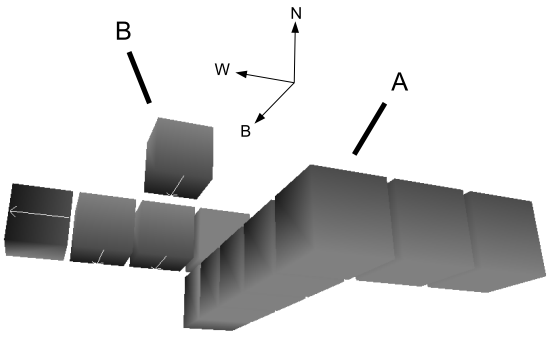


Figure 6.50. The TR mechanism.

Figure 6.51 shows the Petri net for TR.

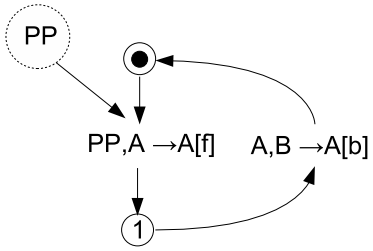


Figure 6.51. Petri net for TR.

6.3.4.10 Program Plane PP

Figure 6.52 shows a portion of the program plane. This corresponds to a single state in the abstract Turing machine.

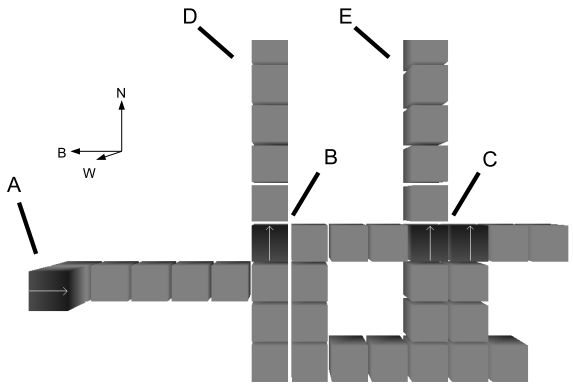


Figure 6.52. The PP mechanism.

The part labelled A is the part that activates the trigger mechanism TR which in turn activates the sequencer SQ when a state position is reached. The pair of parts labelled B correspond to the two bits of information needed to tell the Turing machine what to do when it encounters a 0 on the data tape DT. Similarly, the pair of parts labelled C tell the Turing machine what to do when it encounters a 1 on the data tape. For B and C, a part p with $p.main$ pointing West towards the body of the Turing machine and $p.secondary$ pointing North represents 1, and a part in any other legal orientation represents 0. In both B and C, the first (i.e. back-most) of the two bits of information tells the machine whether to move DT (1) or not (0). The second of the two bits tells the machine whether to write a 1 or a 0 onto DT. The paths labelled D and E are the paths that the plane moving mechanism PM tracks when leaving this state for another state. Which path PM

follows depends on whether a 0 or a 1 was encountered on DT.

Figure 6.53 shows the corner of a track in PP. This figure should be examined in conjunction with the Petri net for PM in figure 6.49 to see how PM and PP negotiate a corner.

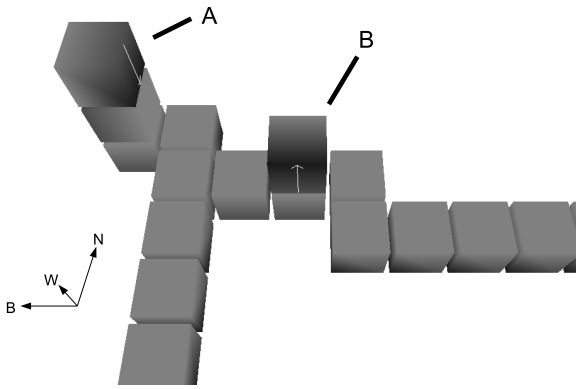


Figure 6.53. The PP mechanism.

Figures 6.54 and 6.55 show partial Petri nets for the program plane PP. (The complete Petri net would be large and not instructive, so is omitted).

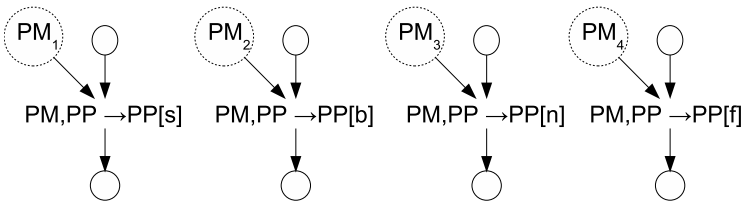


Figure 6.54. Petri net for PP. PP being moved by PM.

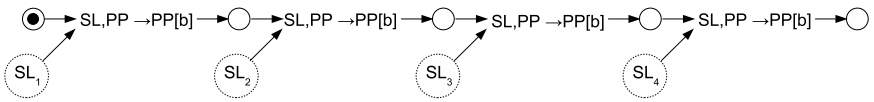


Figure 6.55. Petri net for PP showing the action of SL on PP.

6.3.5 Summary

As it stands, this design would not be suitable for use in the SRPC of Chapter 5 because it contains mechanisms made from disconnected structures which would fall apart when the SRPC moved. It also takes a very long time to carry out a simple computation.

An SRPC could not be made using only the part type used in this section because there is no means of joining parts together. This could be solved by altering the laws of the CBlocks3D environment so that if two parts are pushed in opposite directions they automatically become connected. With this modification, whether an SRPC could be made using a single part type is an open question.

Chapter 7

Discussion and Conclusion

In Chapter 2 it was shown that previous work on self-replicating programmable constructors (SRPC) has focused either on highly abstract systems in cellular automata environments, or on simplified physical systems with either complex parts or a limited constructional capability.

Chapter 3 demonstrated the feasibility of simulating self-replicating systems in kinematic environments that lie in the largely unexplored region between cellular automata and physical systems. Work from Chapter 3 has been published in references [60] and [61].

The CBlocks3D environment described in Chapter 4 is a realisation of the kinematic system proposed by von Neumann [71]. This environment can be reasoned about with the same ease as cellular automata, but also supports concepts of motion and connectivity.

Although the CBlocks3D environment has features which distinguish it from cellular automata, section 4.4.1 shows how the *Hashlife* algorithm [25] for efficiently simulating certain classes of cellular automata patterns can be adapted and used to simulate structures in the CBlocks3D environment.

The design given in Chapter 5 is the first demonstration that it is possible to devise an SRPC in a kinematic simulation environment that supports part types designed to be as simple as possible. The problem of classifying an unknown part using mechanisms made only from the types of part to be classified has been solved in this environment and is published in reference [64].

The SRPC has been simulated in full for a single replication cycle and the resulting child machine has also been simulated for a single replication cycle. The child machine and the grandchild machine were compared and found to be identical. After completing a replication cycle, the SRPC loops back to the beginning of its construction program and

begins constructing another child machine.

A control unit architecture supporting the execution of a linear sequence of instructions, with subroutines for frequently used subsequences, was shown to be sufficient for enabling the SRPC to contain within its memory the instructions needed for constructing a duplicate machine.

The number of parts in the machine (59,615) and the complexity of the set of part types used provide some kind of upper bound for machine size and part complexity for machines with equivalent functionality in a three dimensional discrete kinematic space. Chapter 6 outlines an approach by which the number of part types might be further reduced by using kinematic information processing mechanisms. Work from Chapter 6 has been published in references [62] and [63].

Figure 7.1 shows the graph introduced at the beginning of Chapter 2. The large blue star on this graph shows how the design from Chapter 5 moves closer towards region G from the direction of abstract automata towards physical realism.

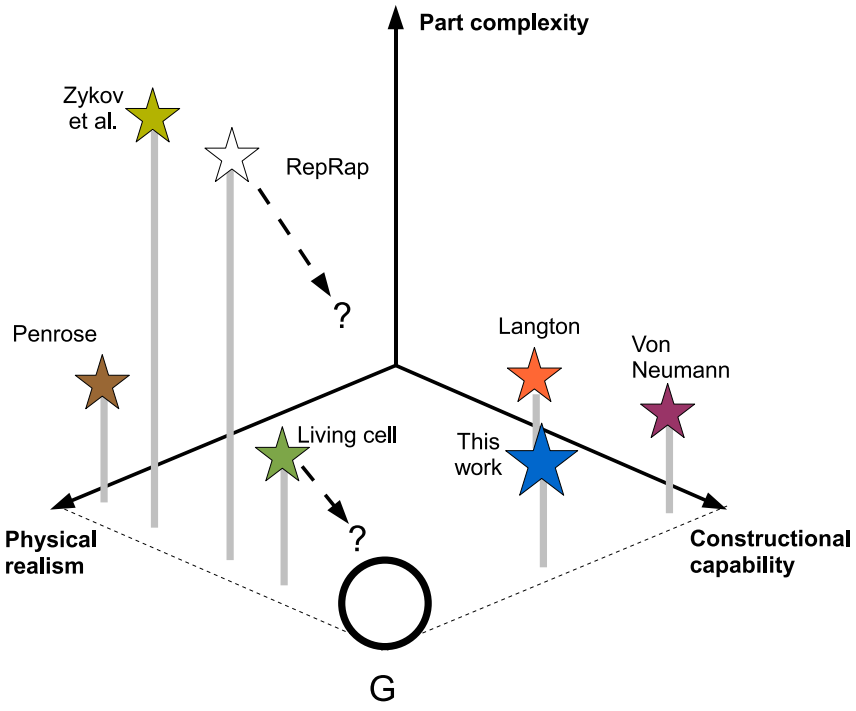


Figure 7.1. Graph showing this work in relation to region G.

7.1 Limitations

This section examines some of the limitations of the work described in this thesis and suggests ways that these limitations can be overcome. In many cases these limitations lead to further unanswered questions that may be the subject of future work.

7.1.1 Construction program design

The sequence of instructions that the SRPC of Chapter 5 executes in order to construct another machine was written manually. For complex constructions requiring lengthy construction programs this is a laborious process. It may be possible to write software to fully or partially automate the process. This software would ideally take a description of a machine to be constructed and translate it into the sequence of construction operations required to construct the machine.

For structures that are smaller than the operating envelope of the *construction arm*, that have no special connectivity requirements and which do not contain more parts than can be stored in all of the *storage mechanisms*, this translation operation should be straightforward. For structures that are larger than this, or that have complex connectivity requirements (such as the *call register* of Figure 5.37 and the *construction arm* of section 5.7), and also for structures that require complex signal patterns (such as the *memory address counter* of Figure 5.33), or where the number of any single part type exceeds the capacity of a *storage mechanism*, the translation operation would be more difficult.

If the general translation task proves to be intractable then some kind of compromise could be reached in which a piece of software produces a construction program that ignores any connectivity issues, leaves all signal loops empty and ignores any requirement to collect new parts when a *storage mechanism* is empty. A human programmer could then refine the program, taking these factors into account.

Just as compilers for programming languages can optimize the code that they produce, a program for automatically translating descriptions of structures into a sequence of construction operations could carry out some optimizations automatically. For example, it could decide which subsequences should be chosen as subroutines so as to minimise the total number of instructions. It could also work out, for a given construction program, whether the pattern of movement and placement followed by the construction head can be rewritten so as to increase the number of common subsequences within the construction program.

7.1.2 Arrangement of parts in the machine's environment

The SRPC described in Chapter 5 requires that all of the parts that it will collect from its environment be laid out in a line to the $WEST$ of the input orifice of the *detect* mechanism. If there are any other parts to the $WEST$ of the body of the machine that are not on this line then they could enter the body of the machine and interfere with its operation.

To prevent parts from entering the machine a barrier could be placed at the $WEST$ end of the machine. This would not fully solve the problem of parts from the environment disrupting the machine, however, because interactions within a build-up of random parts against this barrier could cause unwanted motion to occur which could disrupt the operation of the machine. The barrier could be constructed from a plane of *nor* parts which activate a plane of *slide* parts that would move any parts encountered to the edge of the barrier so that they would not build up. Alternatively the activated *slide* parts on the barrier could be arranged in such a way as to move any parts encountered into the input orifice. Care would have to be taken to avoid situations in which the *slide* parts on the barrier attempted to move several parts into the input orifice at the same time.

7.1.3 Finite machine size

Unlike the automata of von Neumann [70], Codd [14] and others in which the memory or instruction tape of the automata are infinitely extensible, the SRPC described in Chapter 5 has a finite memory and therefore an upper limit on the size of the constructions that it can construct.

Given that it was shown in Chapter 6 that a Turing machine can be embedded within the CBlocks3D environment, it seems reasonable to suppose that an SRPC with an arbitrary sized memory could be implemented in the CBlocks3D environment.

The part storage areas of section 5.5 have a finite capacity, but there seems to be no reason why they could not be redesigned so as to be infinitely extensible. If this were done then the machine would never have to discard any parts that it encountered.

7.1.4 Synchronicity

The CBlocks3D environment is synchronous: all parts in the environment update at the same instant. Many physical systems are not naturally synchronous. If synchronous behaviour is required in a physical system then often a global clock must somehow be distributed to every part of the system. For example, if synchronous behaviour is required

in a digital electronic circuit then every component that needs to be synchronised must be fed a clock signal through a wire dedicated to this purpose.

Conventional cellular automata are also synchronous. Beginning with [45] some researchers have explored how cellular automata behave if the assumptions about when individual cells update their state are altered or removed altogether.

Similar investigations could be carried out for kinematic environments. However, kinematic environments in which the far end of a structure moves instantaneously when the near end is pushed have a natural type of synchronicity that is an essential feature of the system rather than a constraint that can be relaxed.

7.1.5 Construction arm limitations

During the early design of the SRPC described in Chapter 5 it was necessary to decide whether to have a *construction head* fixed to the machine and move the whole machine around in space in order to position the *construction head* (as was done in the machine described in section 3.1.3) or whether to allow the *construction head* to move independently of the rest of the machine. The latter option was chosen for the following reasons:

- It avoids potential movement conflicts. In the CBlocks3D environment all motion is absolute and so a moving mechanism cannot have any internal movement as it moves. If the whole SRPC moved in order to move the *construction head* then it would be necessary to make sure that no mechanisms containing moving parts were active at the time that the machine moved.

- It simplifies part collecting. If the machine does not move around when it is constructing, then parts can be lined up in front of the input orifice of the *detect* mechanism knowing that the position of this orifice on the $NORTH\text{-}SOUTH$ and $FRONT\text{-}BACK$ axes will not change. If the whole machine were able to move around then it would be necessary to work out in advance where the machine would be when it next needed to collect parts from its environment in order to make sure that parts were in the correct location.

- Initially it was hoped that the machine would eventually have an extruding brick type architecture as described in section 14.3.2 of reference [19] in which the child machine emerges from an orifice in the parent machine. In order for this to be possible the *construction head* must be able to move internally within the parent machine. Implementing this type of architecture proved to be beyond the scope of

this thesis but the *construction head* arrangement had already been fixed by the time that this was realized.

None of these reasons are insurmountable. Having a fixed *construction head* and a moving machine would simplify the design slightly and would probably make the translation task described in section 7.1.1 easier.

7.2 Future work

7.2.1 Variations on the environment

Chapter 3 described two environments supporting similar types of part. These environments differed mainly in the type of space that the parts were embedded in. It is interesting to consider whether the result of Chapter 5 can be recapitulated in a continuous space environment. Problems of positioning and orientation are likely to be more difficult in a continuous space environment, as was found for the environment of section 3.2.

7.2.2 Kinematic computing

As was discussed in Chapter 6, the embodiment of computing machines in mechanical or kinematic environments is a relatively unexplored area of research. Traditionally the process of implementing a computing device in the physical world begins with an abstract logical description of the device to be implemented. Physical processes are then sought that can be made to behave in the same way as the abstract description. Stepney [59] advocates a different approach in which systems suspected of having behaviour rich enough to support complex dependencies between inputs and outputs are examined for intrinsic behaviour that can be used to directly carry out a computation.

The two computing schemes described in Chapter 6 show that kinematic automata are rich enough to be used for this kind of investigation.

7.2.3 Physical implementation

Although it was stated in section 3.3 that this thesis would not attempt to make a simulation environment that relates directly to the physical world, one of the goals that this research is motivated by is that of making a physical SRPC. Therefore it is natural to consider whether the part types used in the CBlocks3D model can actually be built and whether they can be made to behave in the way described in Chapter 4.

There are some physical environments in which at least some of the functions listed in Table 3.6 on page 63 can be implemented very simply. Below are speculations on two different systems.

7.2.3.1 Components mounted on floating discs

Consider discs floating on the surface of a liquid with an electronic component mounted on each disc. Metal terminals around the edges of discs serve to connect components together. Capillary forces cause discs to attract one another and stick together. The discs will tend to coalesce into hexagonally packed arrangements. Adjacent discs behave as though there is a bond between them. Terminals in contact form a conductive path between two discs along which signals can propagate. Circuits can be made from collections of discs. A strong magnetic field could permeate the environment so that inductors could be used to generate forces on discs which could be harnessed to propel structures across the surface of the liquid.

7.2.3.2 Fluidic logic

A single piece of moulded plastic with no solid moving parts can function as a fluidic logic gate. The working fluid itself is the only moving part. The shape of the plastic causes the fluid inside it to become a logic gate when the correct conditions are applied. An introduction to fluidic logic can be found in reference [43].

It may be possible to make fluidic logic components that sit above an air-table, where the jets of air are arranged in a square lattice. The jets of air could serve both to keep the components aligned to the lattice and to drive the logic within them.

As well as being able to carry out Boolean operations, it is possible to imagine that jets of air under the control of some components could cause other components on the air-table to conditionally attract or repel each other. It is conceivable that all of the functions listed in table 3.6 on page 63 could be implemented using fluidic components.

7.3 Concluding remarks

An intangible result of this work, which is a consequence of working on the systems described in this thesis, is the experience and insight that I have gained about SRPCs and about kinematic environments. I feel fortunate to have the same enthusiasm for the subject that I had when I began this work, and I hope to use the insight that I have gained to continue this research in some of the directions outlined above.

Appendix A

CBlocks3D Implementation

Below is a C++ program for simulating the CBlocks3D environment. The algorithm is based on ideas from the *Hashlife* algorithm described by Gosper in [25].

```
#include <assert.h>

typedef enum {empty,wire,nor,slide,fuse,unfuse,rotate} blockType;

typedef enum {north,east,south,west,front,back,none} direction;

typedef enum {northmask = 1, eastmask = 2, southmask = 4, westmask = 8,
    frontmask = 16, backmask = 32} directionMasks;

typedef struct
{
  int x,y,z;
} vector;

#define BLOCK_BITS 18
// The bits used in block are as follows:
// 0-2 block type
// 3-7 orientation
// 8 output
// 9-14 connectivity
// 15-17 move propagation direction (direction+1)
//
// 0 represents an empty cell
typedef unsigned block;
```

```
#define EMPTY_FLAG    1
#define NOTPROP_FLAG  2
#define PROPSTATIC_FLAG 4
#define NOTACTIVE_FLAG  8
#define VISITED_FLAG  128

#define CALCINDEX_CALC   0
#define CALCINDEX_PROP   1
#define CALCINDEX_DOMOVE 2
#define CALCINDEX_ACTIVE 3
#define CALCINDEX_CALC2   4

typedef struct otNode
{
  unsigned char level,flags;
  otNode *calc[5];

  union
  {
    block leaf;
    struct otNode *children[2][2][2];
  } u;

  struct otNode *next; // for use in hash and also in freeList
} otNode;

typedef otNode *(*calcFuncPtr)(int x, int y, int z, otNode *o);

vector dirVectorLookup[6] =
{
  {0,1,0},
  {1,0,0},
  {0,-1,0},
  {-1,0,0},
  {0,0,1},
  {0,0,-1},
};

direction opposite[6] =
```

```
{
  south,west,north,east,back,front
};

// For looking up what frontDir should be, given northDir and eastDir
direction frontDirLookup[6][6] =
{
  {none,front,none,back,west,east},
  {back,none,front,none,north,south},
  {none,back,none,front,east,west},
  {front,none,back,none,south,north},
  {east,south,west,north,none,none},
  {west,north,east,south,none,none},
};

// given a 5 bit orientation value, this table stores the
// northDir, eastDir and frontDir
direction dirLookup[24][3] =
{
  {north,east,front}, // 0
  {north,west,back},
  {north,front,west},
  {north,back,east},
  {east,north,back}, // 4
  {east,south,front},
  {east,front,north},
  {east,back,south},
  {south,east,back}, //8
  {south,west,front},
  {south,front,east},
  {south,back,west},
  {west,north,front}, // 12
  {west,south,back},
  {west,front,south},
  {west,back,north},
  {front,north,east}, // 16
  {front,east,south},
  {front,south,west},
  {front,west,north},
  {back,north,west}, // 20
```

```
  {back,east,north},
  {back,south,east},
  {back,west,south},
};
unsigned char makeOrientation[6][6] =
{
  {0,0,0,1,2,3},
  {4,0,5,0,6,7},
  {0,8,0,9,10,11},
  {12,0,13,0,14,15},
  {16,17,18,19,0,0},
  {20,21,22,23,0,0},
};
unsigned char rotateLookup[24][6];

otNode *eg;

#define ALLOC_TABLE_SIZE 45000000
#define GC_THRESHHOLD 100000

otNode *allocTable;

otNode *freeList;
unsigned freeCount;

// First 100 elements of hash table are empty nodes
#define HASH_SIZE 44999991
#define EMPTY_LIST_SIZE 100

otNode **hashTable;

unsigned lookupOrientationValue(direction p, direction s)
{
  for(unsigned i = 0; i<24; i++)
  {
    if (dirLookup[i][0] == p && dirLookup[i][1] == s)
    {
      return i;
    }
  }
```

```
  return 0;
}

unsigned getBlockType(block b) {return b&0x00000007;}
void setBlockType(block *b, unsigned tp) {*b = (*b & ~0x00000007)|tp;}

unsigned getOrientation(block b) {return (b&0x000000f8)>>3;}
void setOrientation(block *b, unsigned orientation)
{*b = (*b & ~0x000000f8)|(orientation<<3);}

unsigned getOutput(block b) {return (b&0x00000100)>>8;}
void setOutput(block *b, unsigned output)
{*b = (*b & ~0x00000100)|(output<<8);}

unsigned getConnect(block b) {return (b&0x00007e00)>>9;}
void setConnect(block *b, unsigned connect)
{*b = (*b & ~0x00007e00)|(connect<<9);}

unsigned getMovedir(block b) {return (b&0x00038000)>>15;}
void setMovedir(block *b, unsigned d) {*b= (*b & ~0x00038000)|(d<<15);}

void alterConnect(block *b, unsigned d, bool c)
{
  if (c)
    *b = *b | (1<<(d+9));
  else
    *b = *b & ~(1<<(d+9));
}

void markTree(otNode *o)
{
  if (o && !(o->flags & VISITED_FLAG))
  {
    o->flags |= VISITED_FLAG;

    if (o->level != 0)
    {
      markTree(o->calc[0]);
      markTree(o->calc[1]);
```

```
      markTree(o->calc[2]);
      markTree(o->calc[3]);
      markTree(o->calc[4]);

      markTree(o->u.children[0][0][0]);
      markTree(o->u.children[0][0][1]);
      markTree(o->u.children[0][1][0]);
      markTree(o->u.children[0][1][1]);
      markTree(o->u.children[1][0][0]);
      markTree(o->u.children[1][0][1]);
      markTree(o->u.children[1][1][0]);
      markTree(o->u.children[1][1][1]);
    }
  }
}

void removeUnmarked(void)
{
  for(unsigned i = 0; i<HASH_SIZE; i++)
  {
    otNode **ptrPtr = &(hashTable[i]);
    otNode *ptr = *ptrPtr;

    while(ptr)
    {
      if (ptr->flags & VISITED_FLAG)
      {
        ptr->flags &= ~VISITED_FLAG;

        ptrPtr = &(ptr->next);
      }
      else
      {
        *ptrPtr = ptr->next;

        destroyNode(ptr);
      }

      ptr = *ptrPtr;
    }
```

```
  }
}

// Garbage collect by traversing the tree, including all calc nodes, marking
// everything as visited, then going through the hash table and removing
// anything that is not visited
void garbageCollect(otNode *o)
{
  markTree(o);
  removeUnmarked();
}

void initTables(void)
{
    allocTable = new otNode[ALLOC_TABLE_SIZE];
    hashTable = new otNode *[HASH_SIZE];

  for(unsigned i = 0; i<HASH_SIZE; i++)
  {
    hashTable[i] = NULL;
  }

  for(unsigned i = 0; i<ALLOC_TABLE_SIZE-1; i++)
  {
    allocTable[i].next = &(allocTable[i+1]);
  }

  allocTable[ALLOC_TABLE_SIZE-1].next = NULL;

  freeList = &allocTable[0];
  freeCount = ALLOC_TABLE_SIZE;

  for(unsigned i = 0; i<24; i++)
  {
    for(unsigned a = 0; a<6; a++)
    {
      if (a == dirLookup[i][0])
      {
        rotateLookup[i][a] =
                  makeOrientation[dirLookup[i][0]][opposite[dirLookup[i][2]]];
```

```
      }
      else if (a==opposite[dirLookup[i][0]])
      {
        rotateLookup[i][a] = makeOrientation[dirLookup[i][0]][dirLookup[i][2]];
      }
      else if (a == dirLookup[i][1])
      {
        rotateLookup[i][a] = makeOrientation[dirLookup[i][2]][dirLookup[i][1]];
      }
      else if (a==opposite[dirLookup[i][1]])
      {
        rotateLookup[i][a] =
          makeOrientation[opposite[dirLookup[i][2]]][dirLookup[i][1]];
      }
      else if (a == dirLookup[i][2])
      {
        rotateLookup[i][a] =
          makeOrientation[opposite[dirLookup[i][1]]][dirLookup[i][0]];
      }
      else if (a==opposite[dirLookup[i][2]])
      {
        rotateLookup[i][a] =
          makeOrientation[dirLookup[i][1]][opposite[dirLookup[i][0]]];
      }
    }
  }
}

unsigned hashFunction(otNode *o)
{
  if (o->flags & EMPTY_FLAG)
  {
    return o->level;
  }
  else if (o->level == 0)
  {
    return o->u.leaf+EMPTY_LIST_SIZE+
      ((o->flags&PROPSTATIC_FLAG)?(1<<BLOCK_BITS):0);
  }
  else
```

```
  {
    return ((((unsigned)(o->u.children[0][0][0]))<<0) +
        (((unsigned)(o->u.children[0][0][1]))<<1) +
        (((unsigned)(o->u.children[0][1][0]))<<2) +
        (((unsigned)(o->u.children[0][1][1]))<<3) +
        (((unsigned)(o->u.children[1][0][0]))<<4) +
        (((unsigned)(o->u.children[1][0][1]))<<5) +
        (((unsigned)(o->u.children[1][1][0]))<<6) +
        (((unsigned)(o->u.children[1][1][1]))<<7))
        %(HASH_SIZE-EMPTY_LIST_SIZE-(1<<(BLOCK_BITS+1)))+
        EMPTY_LIST_SIZE+(1<<(BLOCK_BITS+1));
  }
}

unsigned hashFunctionBlock(block b, unsigned char flags)
{
  return b+EMPTY_LIST_SIZE+((flags&PROPSTATIC_FLAG)?(1<<BLOCK_BITS):0);
}

otNode *addToHash(otNode *o, unsigned index)
{
  o->next = hashTable[index];
  return hashTable[index] = o;
}

bool compare(otNode *o1, otNode *o2)
{
  if (o1->level == o2->level)
  {
    if (o1->level == 0)
    {
      return o1->u.leaf == o2->u.leaf &&
        ((o1->flags & PROPSTATIC_FLAG) == (o2->flags & PROPSTATIC_FLAG));
    }
    else if ((o1->flags & EMPTY_FLAG) || (o2->flags & EMPTY_FLAG))
    {
      return (o1->flags & EMPTY_FLAG) && (o2->flags & EMPTY_FLAG);
    }
    else
    {
```

```
      return !memcmp(&o1->u,&o2->u,8*sizeof(otNode *));
    }
  }
  else
  {
    return false;
  }
}

otNode *hashLookup(otNode *o, unsigned index)
{
  otNode *ptr = hashTable[index];

  while(ptr)
  {
    if (compare(o,ptr))
    {
      return ptr;
    }

    ptr = ptr->next;
  }

  return NULL;
}

#define hashLookupBlock(b,flags,index) hashTable[index];

// Check whether a newly created node is already in the hash
// if so then destroy o and return the one from the hash
// otherwise add o to the hash and return it
otNode *hashCheck(otNode *o)
{
  unsigned index = hashFunction(o);

  otNode *tmp = hashLookup(o,index);

  if (tmp != NULL)
  {
    destroyNode(o);
```

```
    return tmp;
  }
  else
  {
    return addToHash(o,index);
  }
}

otNode *hashCheckBlock(block b, unsigned char flags)
{
  unsigned index = hashFunctionBlock(b,flags);

  otNode *tmp = hashLookupBlock(b,flags,index);

  if (tmp != NULL)
  {
    return tmp;
  }
  else
  {
    otNode *o = createNode(0,false,false);
    o->u.leaf = b;
    o->flags = flags;

    return addToHash(o,index);
  }
}

block getBlockAtLocation(otNode *o, int x, int y, int z)
{
  if (o->flags & EMPTY_FLAG)
    return 0;

  switch(o->level)
  {
  case 0:
    return o->u.leaf;
  case 1:
    return o->u.children[x][y][z]->u.leaf;
  case 2:
```

```
    return o->u.children[x/2][y/2][z/2]->u.children[x&1][y&1][z&1]->u.leaf;
  default:
    {
      int d = (1<<(o->level-1));

      int xo = !(x<d);
      int yo = !(y<d);
      int zo = !(z<d);

      return getBlockAtLocation(
        o->u.children[xo][yo][zo],
        x%d,
        y%d,
        z%d);
    }
    break;
  }
}

#define GETBLOCK_L2(o,x,y,z) \
  (o->u.children[(x)>>1][(y)>>1][(z)>>1]-> \
   u.children[(x)&1][(y)&1][(z)&1]->u.leaf)
#define GETFLAGS_L2(o,x,y,z) \
  (o->u.children[(x)>>1][(y)>>1][(z)>>1]-> \
   u.children[(x)&1][(y)&1][(z)&1]->flags)

// Make a mutable copy of the subtree in which the block resides, then
// make the change
otNode *setBlockAtLocation(otNode *o, int x, int y, int z, block b)
{
  otNode *rval = duplicateNode(o);
  rval->calc[0] = rval->calc[1] = rval->calc[2] = rval->calc[3] = NULL;
  rval->next = NULL;

  switch(o->level)
  {
  case 0:
    rval->u.leaf = b;
    if (getBlockType(b)==empty)
      rval->flags = EMPTY_FLAG|NOTPROP_FLAG|PROPSTATIC_FLAG|NOTACTIVE_FLAG;
```

```
    else
      rval->flags = NOTPROP_FLAG|PROPSTATIC_FLAG|NOTACTIVE_FLAG;
    break;
  default:
    {
      int d = (1<<(o->level-1));

      int xo = ((x<d)?0:1);
      int yo = ((y<d)?0:1);
      int zo = ((z<d)?0:1);

      rval->u.children[xo][yo][zo] = setBlockAtLocation(
            rval->u.children[xo][yo][zo],
        x-xo*d,
        y-yo*d,
        z-zo*d,
        b);

      if (getBlockType(b)!=empty)
        rval->flags &= ~EMPTY_FLAG;
    }
    break;
  }

  return hashCheck(rval);
}

void destroyNode(otNode *o)
{
  o->next = freeList;
  freeList = o;
  freeCount++;
}

otNode *getEmptyNode(unsigned level);

otNode *createNode(unsigned level, bool populated, bool lookupEmpty)
{
  otNode *rval = freeList;
  freeList = freeList->next;
```

```
  freeCount--;

  rval->level = level;
  rval->calc[0] = rval->calc[1] = rval->calc[2] =
    rval->calc[3] = rval->calc[4] = NULL;
  rval->flags = EMPTY_FLAG|NOTPROP_FLAG|PROPSTATIC_FLAG|NOTACTIVE_FLAG;
  rval->next = NULL;

  switch(level)
  {
  case 0:
    rval->u.leaf = 0;
    break;
  default:
    {
      for(int i = 0; i<2; i++)
        for(int j = 0; j<2; j++)
          for(int k = 0; k<2; k++)
          {
            if (populated)
            {
              if (lookupEmpty)
              {
                rval->u.children[i][j][k] = getEmptyNode(rval->level-1);
              }
              else
              {
                rval->u.children[i][j][k] =
                  createNode(rval->level-1,populated,lookupEmpty);
              }
            }
            else
            {
              rval->u.children[i][j][k] = NULL;
            }
          }
    }
    break;
  }
```

```
  return rval;
}

// create a duplicate node with everything the same
otNode *duplicateNode(otNode *o)
{
  otNode *rval = createNode(o->level,false,false);

  *rval = *o;

  return rval;
}

otNode *getEmptyNode(unsigned level)
{
  if (hashTable[level] != NULL)
    return hashTable[level];
  else
  {
    otNode *rval = createNode(level,true,true);
    hashTable[level] = rval;

    return rval;
  }
}

// The PROPSTATIC_FLAG is used in a different way in leaf nodes to other flags
// In leaf nodes, it is used simply to return a value from a process function
otNode *processBlockDoMove(int x, int y, int z, otNode *o)
{
  block b = GETBLOCK_L2(o,x,y,z);
  unsigned char flags;

  if (getMovedir(b))
  {
    b = 0;
  }

  for(unsigned i = 0; i<6; i++)
  {
```

```
    vector p = dirVectorLookup[i];

    block bn = GETBLOCK_L2(o,x+p.x,y+p.y,z+p.z);

    if (getMovedir(bn))
    {
      if (getMovedir(bn)-1 == opposite[i])
      {
        b = bn;
        setMovedir(&b,0);
      }
    }
  }

  flags = NOTPROP_FLAG|NOTACTIVE_FLAG|PROPSTATIC_FLAG;

  if (b==0)
  {
    flags |= EMPTY_FLAG;
  }

  return hashCheckBlock(b,flags);
}

otNode *processBlockProp(int x, int y, int z, otNode *o)
{
  block b = GETBLOCK_L2(o,x,y,z);
  unsigned char flags;

  if (b==0)
  {
    flags = EMPTY_FLAG|NOTPROP_FLAG|NOTACTIVE_FLAG|PROPSTATIC_FLAG;
    return hashCheckBlock(b,flags);
  }

  if (getMovedir(b))
  {
    flags = NOTACTIVE_FLAG|PROPSTATIC_FLAG;
    return hashCheckBlock(b,flags);
  }
```

```
  for(unsigned i = 0; i<6; i++)
  {
    vector p = dirVectorLookup[i];

    block bn = GETBLOCK_L2(o,x+p.x,y+p.y,z+p.z);

    if (getMovedir(bn))
    {
      if (getMovedir(bn)-1 == opposite[i] ||
        ((getConnect(b) & (1<<i)) && (getConnect(bn) & (1<<opposite[i]))) )
      {
        setMovedir(&b,getMovedir(bn));
      }
    }
  }

  if (getMovedir(b) == 0)
  {
    flags = NOTPROP_FLAG | NOTACTIVE_FLAG | PROPSTATIC_FLAG;
    return hashCheckBlock(b,flags);
  }
  else
  {
    flags = NOTACTIVE_FLAG;
    return hashCheckBlock(b,flags);
  }
}

#define CHECK_FUSEUNFUSE(xo,yo,zo,pri,sec) \
    { \
      block bi = GETBLOCK_L2(o,x+xo,y+yo,z+zo); \
      unsigned orientationi = getOrientation(bi); \
      unsigned tpi = getBlockType(bi); \
      if ((tpi == fuse || tpi == unfuse) && getOutput(bi)) \
      { \
        if (dirLookup[orientationi][0] == pri && \
          dirLookup[orientationi][1] == sec) \
        { \
          vector pc = dirVectorLookup[opposite[sec]]; \
```

```
          block bc = GETBLOCK_L2(o,x+pc.x,y+pc.y,z+pc.z); \
          \
          if (getBlockType(bc) != empty) \
            alterConnect(&b,opposite[sec],tpi==fuse); \
        } \
        if (dirLookup[orientationi][0] == sec && \
          dirLookup[orientationi][1] == pri) \
        { \
          vector pc = dirVectorLookup[opposite[pri]]; \
          block bc = GETBLOCK_L2(o,x+pc.x,y+pc.y,z+pc.z); \
          \
          if (getBlockType(bc) != empty) \
            alterConnect(&b,opposite[pri],tpi==fuse); \
        } \
      } \
    }

otNode *processBlockActive(int x, int y, int z , otNode *o)
{
  block b = GETBLOCK_L2(o,x,y,z);
  unsigned char flags;

  unsigned tp = getBlockType(b);
  unsigned orientation = getOrientation(b);

  if (tp != empty)
  {
    if (tp == slide || tp == fuse || tp == unfuse || tp == rotate)
    {
      setOutput(&b,0);
    }

    flags = NOTPROP_FLAG|NOTACTIVE_FLAG|PROPSTATIC_FLAG;

    for(unsigned i = 0; i<6; i++)
    {
      vector p = dirVectorLookup[i];
      block bi = GETBLOCK_L2(o,x+p.x,y+p.y,z+p.z);

      if (getBlockType(bi) == slide && getOutput(bi) &&
```

```
      dirLookup[getOrientation(bi)][0] == opposite[i])
    {
      setMovedir(&b,dirLookup[getOrientation(bi)][1]+1);
      flags = NOTACTIVE_FLAG;
    }

    if (getBlockType(bi) == rotate && getOutput(bi) &&
      dirLookup[getOrientation(bi)][0] == opposite[i])
    {
      setOrientation(&b,rotateLookup[orientation][opposite[i]]);
      flags = NOTACTIVE_FLAG;
    }

    if ((getBlockType(bi) == fuse || getBlockType(bi) == unfuse)
      && getOutput(bi) && dirLookup[getOrientation(bi)][0] == opposite[i])
    {
      vector pc = dirVectorLookup[dirLookup[getOrientation(bi)][1]];
      block bc = GETBLOCK_L2(o,x+pc.x,y+pc.y,z+pc.z);

      if (getBlockType(bc) != empty)
      {
        alterConnect(&b,dirLookup[getOrientation(bi)][1],
                  getBlockType(bi)==fuse);
      }
    }
  }

  CHECK_FUSEUNFUSE(-1,-1, 0,east,north)
  CHECK_FUSEUNFUSE( 1,-1, 0,west,north)
  CHECK_FUSEUNFUSE(-1, 1, 0,east,south)
  CHECK_FUSEUNFUSE( 1, 1, 0,west,south)
  CHECK_FUSEUNFUSE(-1, 0,-1,east,front)
  CHECK_FUSEUNFUSE( 1, 0,-1,west,front)
  CHECK_FUSEUNFUSE(-1, 0, 1,east,back)
  CHECK_FUSEUNFUSE( 1, 0, 1,west,back)
  CHECK_FUSEUNFUSE( 0,-1,-1,front,north)
  CHECK_FUSEUNFUSE( 0, 1,-1,front,south)
  CHECK_FUSEUNFUSE( 0,-1, 1,back,north)
  CHECK_FUSEUNFUSE( 0, 1, 1,back,south)
}
```

```
  else
  {
    flags = EMPTY_FLAG|NOTPROP_FLAG|PROPSTATIC_FLAG|NOTACTIVE_FLAG;
  }

  return hashCheckBlock(b,flags);
}

otNode *processBlock(int x, int y, int z, otNode *o)
{
  block b = GETBLOCK_L2(o,x,y,z);
  unsigned char flags;

  unsigned tp = getBlockType(b);
  unsigned orientation = getOrientation(b);

  if (!(o->flags & NOTACTIVE_FLAG))
  {
    return hashCheckBlock(b,GETFLAGS_L2(o,x,y,z));
  }

  {
    switch(tp)
    {
    case slide:
    case fuse:
    case unfuse:
    case rotate:
      {
        vector p = dirVectorLookup[dirLookup[orientation][0]];
        block bp = GETBLOCK_L2(o,x+p.x,y+p.y,z+p.z);

        if (getBlockType(bp) != empty)
        {
          block bi = GETBLOCK_L2(o,x-p.x,y-p.y,z-p.z);

          if (getOutput(bi) &&
             ((getBlockType(bi) == wire &&
             dirLookup[getOrientation(bi)][0] !=
             opposite[dirLookup[orientation][0]]) ||
```

```
    (getBlockType(bi) == nor
    && dirLookup[getOrientation(bi)][0] == dirLookup[orientation][0]))
  {
    if (tp != fuse && tp != unfuse)
    {
      setOutput(&b,1);
      flags = NOTPROP_FLAG|PROPSTATIC_FLAG;
    }
    else
    {
      vector po = dirVectorLookup[dirLookup[orientation][1]];
      block bo = GETBLOCK_L2(o,x+p.x+po.x,y+p.y+po.y,z+p.z+po.z);

      if   (getBlockType(bo) != empty &&
        ((tp==fuse &&
          !(getConnect(bp) & (1<<dirLookup[orientation][1]))) ||
        (tp==unfuse &&
          (getConnect(bp) & (1<<dirLookup[orientation][1]))))
        )
      {
        setOutput(&b,1);
        flags = NOTPROP_FLAG|PROPSTATIC_FLAG;
      }
      else
      {
        setOutput(&b,0);
        flags = NOTPROP_FLAG|NOTACTIVE_FLAG|PROPSTATIC_FLAG;
      }
    }
  }
  else
  {
    setOutput(&b,0);
    flags = NOTPROP_FLAG|NOTACTIVE_FLAG|PROPSTATIC_FLAG;
  }
}
else
{
  setOutput(&b,0);
  flags = NOTPROP_FLAG|NOTACTIVE_FLAG|PROPSTATIC_FLAG;
```

```
      }
    }

    break;
  case wire:
    {
      vector p = dirVectorLookup[dirLookup[orientation][0]];
      block bi = GETBLOCK_L2(o,x-p.x,y-p.y,z-p.z);

      if (getOutput(bi) &&
         ((getBlockType(bi) == wire
         && dirLookup[getOrientation(bi)][0] !=
         opposite[dirLookup[orientation][0]]) ||
         (getBlockType(bi) == nor
         && dirLookup[getOrientation(bi)][0] == dirLookup[orientation][0])))
      {
        setOutput(&b,1);
      }
      else
      {
        setOutput(&b,0);
      }
    }

    flags = NOTPROP_FLAG|NOTACTIVE_FLAG|PROPSTATIC_FLAG;
    break;
  case nor:
    {
      unsigned op = 1;

      for(unsigned i = 0; i<6 && op; i++)
      {
        if (i != dirLookup[orientation][0])
        {
          vector p = dirVectorLookup[i];
          block bi = GETBLOCK_L2(o,x+p.x,y+p.y,z+p.z);

          if (getOutput(bi) &&
            ((getBlockType(bi) == wire
            && dirLookup[getOrientation(bi)][0] != i) ||
```

```
              (getBlockType(bi) == nor
              && dirLookup[getOrientation(bi)][0] == opposite[i]))
            {
              op = 0;
            }
          }
        }

        setOutput(&b,op);
      }
      flags = NOTPROP_FLAG|NOTACTIVE_FLAG|PROPSTATIC_FLAG;
      break;
    default:
      flags = EMPTY_FLAG|NOTPROP_FLAG|NOTACTIVE_FLAG|PROPSTATIC_FLAG;
      break;
    }
  }

  return hashCheckBlock(b,flags);
}

otNode *otLevel2NodeCalc(otNode *o, calcFuncPtr calcFunc)
{
  otNode *rval = createNode(1,false,false);

  for(int i = 0; i<2; i++)
    for(int j = 0; j<2; j++)
      for(int k = 0; k<2; k++)
      {
        rval->u.children[i][j][k] = calcFunc(i+1,j+1,k+1,o);

        rval->flags = rval->flags & rval->u.children[i][j][k]->flags;
      }

  return hashCheck(rval);
}

otNode *populateGapNode(otNode *o, int i2, int j2, int k2, int i1, int j1, int k1)
{
  otNode *rval = createNode(o->level-2,false,false);
```

```
  for(int i = 0; i<2; i++)
    for(int j = 0; j<2; j++)
      for(int k = 0; k<2; k++)
      {
        int x = 1+i2*2+i1*2+i;
        int y = 1+j2*2+j1*2+j;
        int z = 1+k2*2+k1*2+k;

        rval->u.children[i][j][k] =
          o->u.children[(x&4)>>2][(y&4)>>2][(z&4)>>2]
           ->u.children[(x&2)>>1][(y&2)>>1][(z&2)>>1]
           ->u.children[x&1][y&1][z&1];

        rval->flags = rval->flags & rval->u.children[i][j][k]->flags;
      }

  return hashCheck(rval);
}

otNode *makeTempChild(otNode *o, int i1, int j1, int k1)
{
  otNode *rval = createNode(o->level-1,false,false);

  for(int i = 0; i<2; i++)
    for(int j = 0; j<2; j++)
      for(int k = 0; k<2; k++)
      {
        rval->u.children[i][j][k] = populateGapNode(o,i1,j1,k1,i,j,k);

        rval->flags = rval->flags & rval->u.children[i][j][k]->flags;
      }

  return hashCheck(rval);
}

otNode *makeIntermed(otNode *o, int i1, int j1, int k1)
{
  otNode *rval = createNode(o->level-1,false,false);
```

```
  for(int i=0;i<2;i++)
    for(int j=0;j<2;j++)
      for(int k=0;k<2;k++)
      {
        int x = i1+i;
        int y = j1+j;
        int z = k1+k;

        rval->u.children[i][j][k] =
          o->u.children[x>>1][y>>1][z>>1]
           ->u.children[x&1][y&1][z&1];

        rval->flags = rval->flags & rval->u.children[i][j][k]->flags;
      }

  return hashCheck(rval);
}

otNode *makeInterChild(otNode **intermed,int i1, int j1, int k1)
{
  otNode *rval = createNode((*intermed)->level+1,false,false);

  for(int i=0;i<2;i++)
    for(int j=0;j<2;j++)
      for(int k=0;k<2;k++)
      {
        int x = i + i1;
        int y = j + j1;
        int z = k + k1;

        rval->u.children[i][j][k] = *(intermed + 3*(3*x+y) + z);

        rval->flags = rval->flags & rval->u.children[i][j][k]->flags;
      }

  return hashCheck(rval);
}

// returns a pointer to a node one level down and centred
otNode *otNodeCalc(otNode *o, calcFuncPtr calcFunc, unsigned calcIndex)
```

```
{
  otNode *rval, *tmp;

  if (o->calc[calcIndex] == NULL)
  {
    if (o->level==2)
    {
      o->calc[calcIndex] = otLevel2NodeCalc(o,calcFunc);
    }
    else if (calcIndex == CALCINDEX_CALC2 && (o->level==3 ||
o->level==4 || o->level==5))
    {
      unsigned nextCalcIndex = (o->level==3)?CALCINDEX_CALC:CALCINDEX_CALC2;
      rval = createNode(o->level-1,false,false);

      otNode *intermed[3][3][3];

      for (int i=0; i<3; i++)
        for(int j=0; j<3; j++)
          for(int k=0; k<3; k++)
          {
            intermed[i][j][k] = makeIntermed(o,i,j,k);
            intermed[i][j][k] =
              otNodeCalc(intermed[i][j][k],calcFunc,nextCalcIndex);
          }

      for(int i=0; i<2; i++)
        for(int j=0; j<2; j++)
          for(int k=0; k<2; k++)
          {
            rval->u.children[i][j][k] = makeInterChild(&intermed[0][0][0],i,j,k);
            rval->u.children[i][j][k] =
              otNodeCalc(rval->u.children[i][j][k],calcFunc,nextCalcIndex);
            rval->flags = rval->flags & rval->u.children[i][j][k]->flags;
          }

      o->calc[calcIndex] = hashCheck(rval);
    }
    else
    {
```

```
      rval = createNode(o->level-1,false,false);

      for (int i=0; i<2; i++)
        for(int j=0; j<2; j++)
          for(int k=0; k<2; k++)
          {
            rval->u.children[i][j][k] = makeTempChild(o,i,j,k);
            rval->u.children[i][j][k] =
              otNodeCalc(rval->u.children[i][j][k],calcFunc,calcIndex);
            rval->flags = rval->flags & rval->u.children[i][j][k]->flags;
          }

      o->calc[calcIndex] = hashCheck(rval);
    }

    return o->calc[calcIndex];
  }
  else
  {
    return o->calc[calcIndex];
  }
}

// determine whether the universe needs to be expanded, and if so then do it
otNode *expandUniverse(otNode *o, bool testNeed)
{
  bool doExpand;

  if (testNeed && o->level >= 2)
  {
    doExpand = false;

    for(int i = 0; i<2; i++)
      for(int j = 0; j<2; j++)
        for(int k = 0; k<2; k++)
          for(int i1 = 0; i1 < 2; i1++)
            for(int j1 = 0; j1<2; j1++)
              for(int k1 = 0; k1<2; k1++)
              {
                if (i1 != 1-i || j1 != 1-j || k1 != 1-k)
```

```
                {
                  if (!o->u.children[i][j][k]->
                                    u.children[i1][j1][k1]->flags & EMPTY_FLAG)
                  {
                    doExpand = true;
                  }
                }
              }
  }
  else
  {
    doExpand = true;
  }

  if (doExpand)
  {
    otNode *rval = createNode(o->level+1,false,false);

    rval->flags = o->flags;

    for(int i = 0; i<2; i++)
      for(int j = 0; j<2; j++)
        for(int k = 0; k<2; k++)
        {
          otNode *tmp = createNode(o->level,true,true);
          tmp->u.children[1-i][1-j][1-k] = o->u.children[i][j][k];
          tmp->flags = o->u.children[i][j][k]->flags;
          rval->u.children[i][j][k] = hashCheck(tmp);
        }

    return hashCheck(rval);
  }
  else
  {
    return o;
  }
}

otNode *singleStep(otNode *eg, unsigned &iters)
{
```

```
bool doneNormal = true;
static int mode = 0;
static int modeCounter = 0;

do
{
  doneNormal = false;

  if (eg->flags & NOTPROP_FLAG)
  {
    if (eg->flags & NOTACTIVE_FLAG)
    {
      otNode *tmp;

      if (mode == 0)
        tmp = expandUniverse(
                  otNodeCalc(eg,processBlock,CALCINDEX_CALC2),false);

      if (mode || !(tmp->flags & NOTACTIVE_FLAG))
      {
        eg = expandUniverse(otNodeCalc(eg,processBlock,CALCINDEX_CALC),false);
        iters+=1;
      }
      else
      {
        eg = tmp;
        iters+=8;
      }

      if (eg->flags & NOTACTIVE_FLAG)
      {
        modeCounter++;
        doneNormal = true;
      }
      else
      {
        modeCounter = 0;
        mode = 1;
      }
```

```
        if (modeCounter > 8) mode = 0;
      }
      else
      {
        eg = expandUniverse(
                  otNodeCalc(eg,processBlockActive,CALCINDEX_ACTIVE),false);

        if (eg->flags & NOTPROP_FLAG)
        {
          doneNormal = true;
        }
      }
    }
    else
    {
      if (eg->flags & PROPSTATIC_FLAG)
      {
        eg = expandUniverse(
                  otNodeCalc(eg,processBlockDoMove,CALCINDEX_DOMOVE),false);
        doneNormal = true;
      }
      else
      {
        eg = expandUniverse(
          expandUniverse(
                    otNodeCalc(eg,processBlockProp,CALCINDEX_PROP),true),false);
      }
    }

    if (freeCount < GC_THRESHHOLD)
    {
      garbageCollect(eg);
    }
  }
  while(!doneNormal);

  return eg;
}
```

Appendix B

Portions of the Construction Program

B.1 Memory Module

This subroutine constructs a single memory module as described in section 5.8. This listing includes all other subroutines that are called. The listing is 232 instruction words long.

```
001 MemoryBlock:
002
003 ORIENT SE
004 SLI
005 ROT
006 FUS
007 UFS
008 CALL NorWest6
009
010 ORIENT BE
011 NOP
012 CALL DelStrip
013 CALL MemSub1
014 ORIENT BE
015 EAST
016 CALL NorWest6
017 SOUTH
018 SOUTH
019 WEST
020
021 CALL MemLayer1_3
022 CALL MemLayer1_3
023 CALL MemLayer1_3
024 CALL MemLayer1_3
025 CALL MemLayer1_3
026
027 ORIENT SE
028 NOP
029 CALL DelStrip
030
031 ORIENT FE
032 NOP
033 CALL DelStrip
034 CALL MemSub1
035 EAST
036 CALL DelWest6
```

```
037
038 CALL East6
039
040 CALL WestWestBack
041
042 NORTH
043 GATHER
044 CALL SouthNorthSouth
045
046 CALL NorthSNSNSNS3
047 CALL NorthSNSNSNS3
048
049 NORTH
050 CALL SouthNorthSouth
051 CALL SouthNorthSouth
052
053 WEST
054 NORTH
055 EAST
056 NORTH
057 CALL SouthNorthSouth
058
059 FrontSouthSouth:
060
061 FRONT
062 SOUTH
063 SOUTH
064
065 RETURN
066
067 MemLayer1_3:
068
069 CALL MemLayer1
070 CALL MemLayer1
071
072 MemLayer1:
073
074 ORIENT SE
075 NOP
076 CALL DelStrip
077
078 ORIENT NE
079 NOP
080 CALL DelStrip
081 ORIENT FE
082 NOP
083 CALL MemSub1
084 EAST
085 CALL DelWest6
086
087 SOUTH
088 SOUTH
089 WEST
090
091 MemLayer2:
092
093 ORIENT SE
094 NOP
095 CALL DelStrip
096
097 ORIENT BE
098 NOP
099 CALL DelStrip
100 CALL MemSub1
101 ORIENT NE
102 EAST
103 CALL DelWest6
104
105 WEST
106 CALL WestBackSouth
107 CALL NorthGatherNorthSouth
108
109 SOUTH
110
111 FrontEastSouth:
112
```

```
113 FRONT
114 EAST
115 SOUTH
116
117 RETURN
118
119 DelStrip:
120
121 FRONT
122 CALL East5
123 BACK
124
125 DelWest6:
126
127 DEL
128 WEST
129 NOP
130 NOP
131
132 DWDWDWDWD:
133
134 DEL
135 WEST
136 NOP
137 NOP
138
139 DWDWDWD:
140
141 DEL
142 WEST
143 NOP
144 NOP
145
146 DWDWD:
147
148 DEL
149 WEST
150 NOP
151 NOP
152
153 DWD:
154
155 DEL
156 WEST
157 DEL
158
159 RETURN
160
161 MemSub1:
162
163 CALL DelStrip
164 CALL DelStrip
165 CALL DelStrip
166 CALL DelStrip
167 CALL DelStrip
168 EAST
169 SOUTH
170 CALL ENN
171 CALL WestWestNorth
172
173 FrontEast5Back:
174
175 CALL FrontEast5
176 BACK
177
178 RETURN
179
180 NorWest6:
181
182 NOR
183 WEST
184 NOP
185 NOP
186
187 NWNWNWNWN:
188
```

```
NOR
WEST
NOP
NOP

NWNWNWN:

NOR
WEST
NOR
WEST

NWN:

NOR
WEST
NOR

RETURN

FrontEast5:

FRONT
CALL East5

RETURN

East6:

CALL East5
EAST

RETURN

East5:

EAST
EAST
NOP
NOP

East3:

EAST
EAST
EAST

RETURN

WestWestNorth:

WEST
WEST
NORTH

RETURN

WestWestBack:

WEST
WEST
BACK

RETURN

NorthSNSNSNS3:

CALL NorthSNSNSNS
CALL NorthSNSNSNS

NorthSNSNSNS:

NORTH
CALL SouthNorthSouth
NORTH
```

```
SouthNorthSouth:

SOUTH
NORTH
SOUTH

RETURN

NorthGatherNorthSouth:

CALL NorthGatherNorth
SOUTH

RETURN

NorthGatherNorth:

NORTH
GATHER
NORTH

RETURN

ENN:

EAST
NORTH
NORTH

RETURN

WestBackSouth:

WEST
BACK
SOUTH

RETURN

END
```

B.2 1 to 4 Pulse Converter

This subroutine constructs a mechanism that converts a single signal pulse into a sequence of 4 signal pulses as described in section 4.3.6. This listing includes all other subroutines that are called. The listing is 64 instruction words long.

```
001 Pulser1to4:
002
003 CALL SE_SDSD
004
005 NORTH
006 NORTH
007 ORIENT FS
008 UFS
009 SOUTH
010 SLI
011
012 NORTH
013
014 FRONT
015 FRONT
016 ORIENT NE
017 DEL
018 CALL SouthFENor
019 NORTH
020
021 BACK
022 BACK
023 FRONT
024 FRONT
025
026 ORIENT BS
027 FUS
028 CALL SouthFENor
029 SOUTH
030 ORIENT BN
031 SLI
032
033 NORTH
034 NORTH
035 BACK
036 BACK
037 ORIENT NE
038
039 DelSouthFEDelSouthSEDel:
040
041 DEL
042 CALL SouthFEDelSouthSEDel
043
044 RETURN
045
046 SouthFENor:
047
048 SOUTH
049 ORIENT FE
050 NOR
051
052 RETURN
053
054 SouthFEDel:
055
056 SOUTH
057 ORIENT FE
058 DEL
059
060 RETURN
061
062 SouthFEDelSouthSEDel:
063
064 CALL SouthFEDel
```

```
065 NOP
066 NOP
067
068 SouthSEDel:
069
070 SOUTH
071 ORIENT SE
072 DEL
073
074 RETURN
075
076 DSD:
077
078 DEL
079 SOUTH
080 DEL
081
082 RETURN
083
084 SE_SDSD:
085
086 ORIENT SE
087 NOP
088 NOP
089 NOP
090
091 SDSD:
092
093 SOUTH
094 CALL DSD
095
096 RETURN
097
098 END
```

Bibliography

[1] Andrew Adamatzky. *Collision Based Computing*. Springer-Verlag, London, 2002.

[2] Bruce Alberts, Dennis Bray, Julian Lewis, Martin Raff, Keith Roberts, and James D. Watson. *Molecular Biology of the Cell, Third Edition*, pages 863–946. Garland Publishing Inc, New York, 1994. These pages describe the cell division cycle.

[3] Bruce Alberts, Dennis Bray, Julian Lewis, Martin Raff, Keith Roberts, and James D. Watson. *Molecular Biology of the Cell, Third Edition*, pages 3–11. Garland Publishing Inc, New York, 1994. These pages discuss the origin of life.

[4] Michael A Arbib. Machines which compute and construct. In *Theories of Abstract Automata*, chapter 10, pages 355–361. Prentice-Hall, Englewood Cliffs, New Jersey, 1969.

[5] Charles Babbage. *Passages from the Life of a Philosopher*, chapter 8. Longman, Green, Longman, Roberts and Green, 1864. URL http://www.fourmilab.ch/babbage/lpae.html (visited on 20th October 2009).

[6] Mark Bedau. *Artificial Life*. MIT Press, Cambridge, Massachusetts.

[7] Jean-Luc Beuchat and Jacques-Olivier Haenni. Von Neumann's 29-state cellular automaton: a hardware implementation. *IEEE Transactions on Education*, 43(3):300–308, 2000.

[8] Nick Bostrom. Existential Risks - Analyzing human extinction scenarios and related hazards. *Journal of Evolution and Technology*, 9, 2002.

[9] Adrian Bowyer. The self-replicating rapid prototyper - Manufacturing for the masses. In *Proceedings of the 8th National Conference on Rapid Design, Prototyping and Manufacturing*. Rapid Prototyping and Manufacturing Association, 2007.

[10] William R Buckley. Signal crossing solutions in von Neumann self-replicating cellular automata. In *Automata 2008*, pages 453–501. Luniver Press, Frome, UK, 2008.

[11] Arthur W Burks. In *Essays on Cellular Automata*, chapter 1. University of Illinois Press, Urbana, Illinois, 1970.

[12] John Byl. Self-reproduction in small cellular automata. *Physica D*, 34:295–299, 1989.

[13] Alexander Graham Cairns-Smith. *Seven Clues to the Origin of Life.* Camridge University Press, Cambridge, 1985.

[14] E F Codd. *Cellular Automata.* Academic Press, New York, 1968.

[15] J Devore and R Hightower. The devore variation of the codd self-replicating computer, 1992. Original work carried out in the early 1970s but never published. Presented at the Third Workshop on Artificial Life, Santa Fe, New Mexico. An implementation of Devore's machine is available at the following URL: http://code.google.com/p/ruletablerepository/wiki/TheRules (visited 8th May 2010).

[16] A K Dewdney. The planiverse project: Then and now. *The Mathematical Intelligencer*, 22(1):46–51, 2000.

[17] K E Drexler. *Engines of Creation: The Coming Era of Nanotechnology.* Anchor Press/Doubleday, New York, 1986.

[18] K E Drexler. Biological and nanomechanical systems: Contrasts in evolutionary capacity. In *Artificial Life*, pages 501–519, Reading, Massachusetts, 1989. Addison-Wesley.

[19] K E Drexler. *Nanosystems.* John Wiley and Sons, New York, 1992.

[20] E Fredkin and T Toffoli. Conservative logic. *Journal of Theoretical Physics*, 21(3,4):219–253, 1982.

[21] Robert A Freitas. A self-replicating interstallar probe. *Journal of the British Interplanetary Society*, 33:251–264, 1980.

[22] Robert A Freitas and William P Gilbreath. Advanced automation for space missions. *NASA Conference Publications*, CP-2255 (N83-15348), 1982. URL http://www.islandone.org/MMSG/aasm/ (visited on 22nd October 2009).

[23] Robert A Freitas and William P Gilbreath. Advanced automation for space missions. *NASA Conference Publications*, CP-2255 (N83-15348):253–257, 1982.

[24] Robert A Freitas and Ralph C Merkle. *Kinematic Self-Replicating Machines.* Landes Bioscience, Georgetown, Texas, 2004.

[25] Gosper. Exploiting regularities in large cellular spaces. *Physica D: Nonlinear Phenomena*, 10(1-2):75–80, January 1984.

[26] Saul Griffith, Dan Goldwater, and J M Jacobson. Robotics: Self-replication from random parts. *Nature*, 437:636, 2005.

[27] Gabor T Herman. On universal computer constructors. *Information Processing Letters*, 2:61–64, 1973.

[28] Tim Hutton. Evolvable self-replicating molecules in an artificial chemistry. *Artificial Life*, 8(4):341–356, 2002.

[29] M C Jewett and G C Church. In vitro integration of ribosomal RNA synthesis, ribosome self-assembly and protein synthesis. *Nature*, 2009.

[30] John Kavanagh and Wendy Hall. *Grand Challenges in Computing Research - GC7: Journeys in Non-Classical Computation.* UK Computing Research Committee, 2008.

[31] John G Kemeny. Man viewed as a machine. *Scientific American*, 192(4):58–67, 1955.

[32] D Kirschner, Y Iwasa, and L Wolpert. *Journal of Theoretical Biology.* Elsevier.

[33] F W Kistermann. Abridged multiplication - the architecture of Wilhelm Schickard's calculating machine of 1623. *Vistas in Astronomy*, 28(1-2):347–353, 1985.

[34] John R Koza. Artificial life: Spontaneous emergence of self-replicating and evolutionary self-improving computer programs. In *Artificial Life III*, pages 225–262, Reading Massachusetts, 1994. Addison-Wesley.

[35] Richard Laing. Some alternative reproductive strategies in artificial molecular machines. *Journal of Theoretical Biology*, (54):64–84, 1975.

[36] Richard Laing. *Automaton self-reference.* PhD thesis, State University of New York, 1978. URL http://hdl.handle.net/2027.42/6125 (visited on 22nd October 2009).

[37] C G Langton. Self-reproduction in cellular automata. *Physica D*, 10:135–144, 1984.

[38] C Y Lee. A Turing machine that prints its own code script. In *Proceedings of the Symposium on the Mathematical Theory of Automata*, pages 155–164, Brooklyn, New York, 1963. Polytechnic Press.

[39] Kiju Lee, Matt Moses, and Gregory S Chirikjian. Robotic self-replication in structured environments: Physical demonstrations and complexity measures. *The International Journal of Robotics Research 2008*, 27(3-4):387–401, 2008. URL http://ijr.sagepub.com/cgi/content/abstract/27/3-4/387 (visited on 22nd October 2009).

[40] Michael T Madigan, John M Martinko, and Jack Parker. *Brock Biology of Microorganisms, Tenth Edition*, chapter 14, page 442. Pearson Prentice-Hall, Upper Saddle River, New Jersey, 2003.

[41] B McMullin. John von Neumann and the evolutionary growth of complexity: Looking backwards, looking forwards... In *Artificial Life VII: Proceedings of the Seventh International Conference*, pages 467–476, Boston Massachusetts, 2000. MIT Press.

[42] Ralph C Merkle. A proposed metabolism for a hydrocarbon assembler. *Nanotechnology*, 8:149–162, 1997.

[43] N M Morris. *Logic Circuits*, pages 180–195. McGraw-Hill, London, 1971.

[44] M Moses. A physical prototype of a self-replicating universal constructor. Master's thesis, Department of Mechanical Engineering, University of New Mexico, 2001.

[45] K Nakamura. Asynchronous cellular automata and their computational ability. *Syst. Comput. Contr.*, 5(5):58–66, 1974.

[46] Renato Nobili. The cellular automata of John von Neumann. http://www.pd.infn.it/ rnobili/wjvn/index.htm (visited on 12th July 2009).

[47] Lionel S Penrose. Self-reproducing machines. *Scientific American*, 200(6):105–114, 1959.

[48] Lionel S Penrose and Roger Penrose. A self-reproducing analogue. *Nature*, 179(4571):1183, 1957.

[49] Umberto Pesavento. An implementatin of von Neumann's self-reproducing machine. *Artificial Life*, 2(4):337–354, 1995.

[50] C A Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, 1962. Information about petri nets is also available at http://en.wikipedia.org/wiki/Petri_net (visited on 22nd October 2009).

[51] Chris Phoenix and Eric Drexler. Safe exponential manufacturing. *Nanotechnology*, 15:869–872, 2004.

[52] James A Reggia, Steven L Armentrout, Hui-Hsien Chou, and Yun Peng. Two-dimensional cellular automaton with either 6 or 8 states per cell and a neighbourhood of either 5 or 9 cells. *Science*, 259:1282–1287, 1993.

[53] R Rojas. Konrad Zuse's legacy: The architecture of the Z1 and Z3. *IEEE Annals of the History of Computing*, 19(2), 1997. URL http://www.epemag.com/zuse/ (visited on 22nd October 2009).

[54] Paul W. K. Rothemund. Folding DNA to create nanoscale shapes and patterns. *Nature*, 440(7082):297–302, Mar 2006.

[55] Moshe Sipper. Fifty years of research on self-replication: An overview. *Artificial Life*, 4(3):237–257, 1998.

[56] Richard Smalley. Of chemistry, love and nanobots. *Scientific American*, 285:76–77, 2001.

[57] Richard Smalley. K Eric Drexler and Richard E Smalley square off on the possibility of molecular assemblers. *Chemical and Engineering News*, 81(48):37–42, 2003.

[58] Arnold Smith, Peter Turney, and Robert Ewaschuk. Self-replicating machines in continuous space with virtual physics. *Artificial Life*, 9:21–40, 2003.

[59] Susan Stepney. The neglected pillar of material computation. *Physica D: Nonlinear Phenomena*, 237(9):1157–1164, July 2008.

[60] William M Stevens. Nodes: An environment for simulating kinematic self-replicating machines. In *Proc. 9th International Conference on the Simulation and Synthesis of Living Systems.*, pages 39–44, Boston Massachusetts, 2004. MIT Press.

[61] William M Stevens. Simulating self-replicating machines. *Journal of Intelligent and Robotic Systems*, 49(2):135–150, 2007.

[62] William M Stevens. Logic circuits in a system of repelling particles. *International Journal of Unconventional Computing*, 4(1):61–77, 2008.

[63] William M Stevens. A kinematic Turing machine. *International Journal of Unconventional Computing*, 5(2):145–163, 2009.

[64] William M Stevens. Parts closure in a kinematic self-replicating programmable constructor. *Artificial Life and Robotics*, 13(2):508–511, 2009.

[65] James W Thatcher. *Universality in the von Neumann cellular model : Technical Report 03105-30-T.* University of Michigan, 1964. URL http://hdl.handle.net/2027.42/7923 (visited 22nd October 2009).

[66] Alan M Turing. On computable numbers with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936. Corrections, ibid, vol 43, pp 544-546, 1937.

[67] Peter Turney and Robert Ewaschuk. Self-replication and self-assembly for manufacturing. *Artificial Life*, 12:411–433, 2006.

[68] John von Neumann. Re-evaluation of the problems of complicated automata - problems of heirarchy and evolution (Fifth Illinois Lecture - December 1949). *Papers of John von Neumann on Computing and Computer Theory*, pages 477–490, 1987.

[69] John von Neumann. Letter to Norbert Wiener from John von Neumann dated November 29th 1946. *Proceedings of Symposia in Applied Mathematics*, 52:506–512, 1997.

[70] John von Neumann and Arthur W Burks. *Theory of Self-Reproducing Automata.* University of Illinois Press, Urbana, Illinois, 1966.

[71] John von Neumann and Arthur W Burks. *Theory of Self-Reproducing Automata*, pages 81–82. University of Illinois Press, Urbana, Illinois, 1966. These pages describe von Neumann's kinematic model.

[72] Norbert Wiener. *Cybernetics, or Control and Communication in the Animal and the Machine.* John Wiley and Sons Inc, New York, 1948.

[73] Edmund Beecher Wilson. Atlas of fertilization and karyokinesis of the ovum. *The American Naturalist*, 29(348):1075–1076, 1895.

[74] Victor Zykov, Efstathios Mytilinaios, Bryant Adams, and Hod Lipson. Self-reproducing machines. *Nature*, 435:163–164, 2005.