The Cosmos Artificial Life System[*]

Tim Taylor
Institute of Perception, Action and Behaviour,
Division of Informatics, University of Edinburgh,
5 Forrest Hill, Edinburgh EH1 2QL, Scotland.
`tim.taylor@ed.ac.uk`

Version 2.0[†]
31 May 1999

[*]This paper is a substantially revised and updated version of DAI Working Paper No. 259 [Taylor 96].
[†]Substantial revisions and addition of new material, 31 May 1999. Version 1.0 published 8 October 1997.

Contents

1 Introduction

This paper describes the 'Cosmos'[1] system, developed to study the evolution of self-replicating (parallel) computer programs. The design is based upon Ray's Tierra system [Ray 91], although there are some significant differences. For a full account of the motivations for building Cosmos, the reader is referred to [Taylor 99].

2 Cosmos Design Philosophy

The basic approach employed in Cosmos to model an evolutionary process is the same as in Tierra. However, many of the design details are different, reflecting the slightly different goals motivating the two systems. One of the original goals of Cosmos was that it should be able to support self-replicating programs with some of the features possessed by simple *cellular* biological organisms, such as mechanisms for communication and response to environmental stimuli (which may potentially promote coevolution between organisms), and mechanisms for regulating the genome (which may promote the evolution of differentiated programs).

Before continuing, clarification should be given of some of the terms that will be used when describing Cosmos. Biological terms will often be used, as these tend to be somewhat more concise than the associated terms relating to computer architectures. While these biological terms suggest the analogy that was in mind when Cosmos was designed, the analogies are certainly not exact; many simplifications and modifications obviously have to be made when designing such a system. With this is mind, the meanings attached to some biological terms in the present context are listed in Table 1.

Term	Meaning in context of Cosmos
Genotype	The instructions that make up a program (the host code within a cell).
Genome	The structure within a program which stores the program's instructions. In the current context, the terms genome and genotype are used more or less interchangeably.
Phenotype	The action (behaviour) of a program as its instructions are being executed.
Organism	A single program, which may be unicellular or multicellular.
Cell	A single process in an organism. This term encompasses the host code and any foreign code that may be present, together with associated working memory, buffers, registers and other structures.
Unicellular	An organism containing a single cell/process (in other words, a serial program).
Multicellular	An organism containing multiple cells/processes (in other words, a parallel program).

Table 1: Definitions of Biologically-Related Terms Used for Describing Cosmos.

Perhaps the most significant difference between Cosmos and Tierra is that programs in Cosmos cannot directly read the code of their neighbours. Cells can only communicate with each other (within or between organisms) by message passing (described in Sections 4.7 and 7.1). Apart from this intercellular communication, each cell only has read, write and execute access within its own cell boundary.

[1]The name Cosmos stands for COmpetitive Self-replicating Multicellular Organisms in Software.

Among the other important differences between Cosmos and Tierra are a number of features in Cosmos intended to encourage the evolution of diversity and complexity[2] in the competing programs, rather than just the optimisation of their ancestral algorithms. The most important of these are the energy token allocation system, described in Sections 4.5 and 6.2, and the regulator system of *promoters* and *repressors* which governs the execution of a program's code, described in Section 4.3. The regulator system is closely linked to the programming language in which the self-replicators are written, introduced in Section 5. Further differences between Cosmos and Tierra are discussed in Section 11.

3 Preliminary Issues

Before going into the details of program representation and behaviour, a few words should be said about some general features of Cosmos.

3.1 Representation of Information

The underlying representation of many of the components of the Cosmos system is the BitString. Four different types of BitString are used: BitStrings, InfoStrings, WritableInfoStrings and EnvironmentalInfoStrings. These are defined as follows:

BitString A vector of binary digits (i.e. a string of 0s and 1s).

InfoString Like a basic BitString, but also has a *type* associated with it (an integer i in the range $0 \leq i \leq 15$), and a pointer to the current read/write position along the string. A string of bits belonging to an InfoString cannot usually be altered after its initial creation—it can only be read. The only exception is that an InfoString may be *mutated*, which entails one or more of its its being flipped at random.

WritableInfoString An InfoString in which the bit string can be written to as well as read from.

EnvironmentalInfoString An InfoString that has an *intensity* (a non-negative real valued number) associated with it.

3.2 Spatial Structure

The shared space in which the organisms reside is a two-dimensional grid, divided into discrete squares.[3] Each cell in the population is associated with a particular square at any given time. This environment can be configured to wrap around, or not to wrap around, in each dimension. More information about the environment in which the cells live is given in Section 6.

3.3 Time Slicing and the Top-Level Algorithm

The Cosmos operating system simulates the parallel execution of a large number of programs. As Cosmos is actually implemented on a serial machine, a form of time slicing is required to achieve this (i.e. at each time slice, a small number of instructions are executed for each program, one at a time). The top-level algorithm that implements this procedure is described in Section 8. At

[2]Many of the design features of Cosmos were intended to promote the evolution of multicellular organisms from unicellular ones.

[3]The system has been designed to deal with arbitrary n-dimensional environments, but the current implementation requires some minor revisions to allow this.

each time slice, it must be decided how many instructions are to be executed for each program. Possibly the most obvious strategy is to execute a fixed number of instructions for each program. However, from an evolutionary point of view, this would introduce selection pressure for small programs because, all else being equal, longer programs would take a larger number of time slices to reproduce. This may or may not be desirable. The decision of how many instructions to run for each program at each time slice is therefore governed by a couple of parameters which can be tuned by the user. Specifically, a program of length L bits is allowed to execute N instructions per time slice, determined by the formula:

$$N = \texttt{et\_value\_constant} * L^{\texttt{et\_value\_power}}$$

N is rounded down to an integer value. This allows considerable flexibility: for example, if `et_value_power` is set to 0.0, then each program executes `et_value_constant` instructions per time slice, regardless of length; if `et_value_power` is set to 1.0, then the allocation is linearly proportional to program length, so evolutionary selection is size-neutral (all else being equal). Further details of time slicing are given in Section 4.5.

3.4 Naming of Organisms

For the purpose of analysis of the system's behaviour, individual organisms are given names according to their genotype. The name is composed of a number followed by a string of (usually four) upper-case alphabetic characters. The number is the length of the genome (expressed as a number of bits) in the organism's initial cell. The character string is a unique identifier for that particular genome. Ancestor organisms inoculated into the system at the start of the run are named with the character string AAAA. If an offspring has an identical genotype to its parent, it will share the same name. If the offspring has a different genotype, then it is given a new name (the operating system keeps track of which names have already been issued, to avoid duplication). For example, the first organism to appear in the system that differs from the inoculated ancestors will be named with the character string AAAB. Should all character strings up to ZZZZ have been issued for organisms of a particular length, an extra A is added to the string (so the next organism of that length with a different genotype to its parent will be named with the character extension AAAAA).

4 The Structure of an Individual Cell

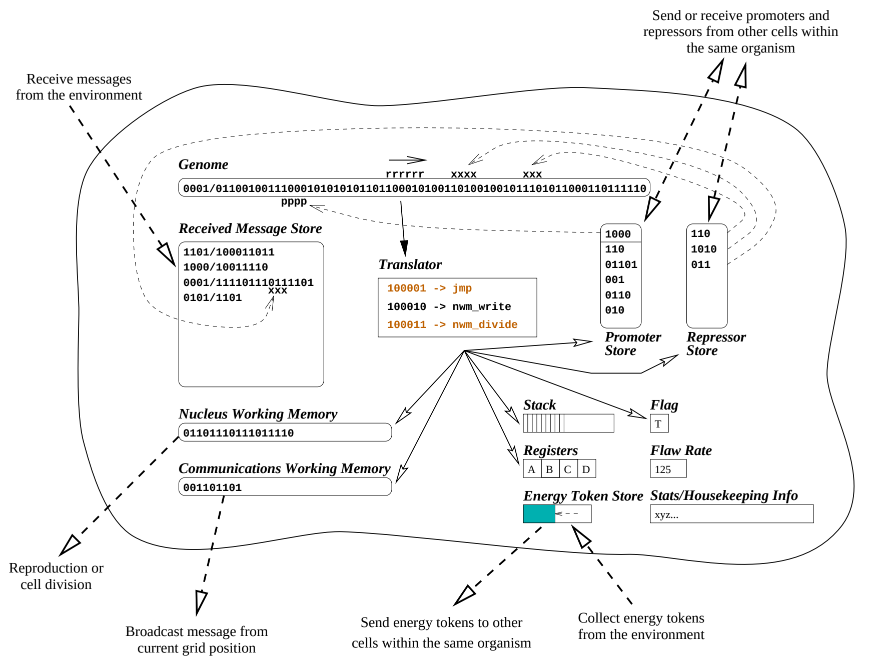
4.1 Overview

The basic structure of a single cell is shown diagrammatically in Figure 1. Each cell is a process running on the (virtual) Cosmos operating system. A cell has its own program code, working memory, stack, registers and various other structures. The major features of the cell are explained in the rest of this section.

4.2 The Genome

The Genome is an InfoString (i.e. a BitString with an associated type), containing encoded instructions that the cell can execute.[4] Which sections of the genome are translated into instructions and executed is determined by the action of promoters and repressors (see Section 4.3). After

[4]As I am usually referring to the contents of the Genome, rather than to the structure itself, when I use the term 'genome', I will use the standard typeface from now on.



Key for Regulator and Translation Symbols:

```
ppp - Promoter Binding Site (marks start of currently active region)
xxx - Repressor Binding Site (translation stops at these sites)
rrr - Current Position of Reading Head
```

Key for Bit Strings:

```
A bit string containing a '/' symbol
is a "typed" bit string (an InfoString).
The digits to the left of the '/'
specify the type of the string.
```

Figure 1: The Structure of a Cell in Cosmos.

a cell has been created, its genome cannot usually be altered, except by the action of mutation (see Section 6.7).

4.3 Regulators: Promoters and Repressors

The translation of the genome is governed by regulators. These are (usually short) BitStrings, and come in two distinct types; promoters and repressors. The cell has a separate store for each of these two types of regulator, and each store can contain a number of regulators of the appropriate type. Regulators may be added to the Promoter Store and Repressor Store in two ways: either by the cell creating a new regulator (by executing an appropriate `reg_create` instruction),[5] or, in the case of a multicellular program (see Section 4.9), by the cell being sent a regulator from a neighbouring cell. A cell can also remove regulators from its Promoter Store and Repressor Store, by executing an appropriate `reg_destroy` instruction.

4.3.1 Promoters and the Promoter Store

The Promoter Store is an ordered list of promoters. Only the promoter currently at the top of the list is active at any given time. The active promoter specifies the position along the genome[6] at which translation will begin. When a new promoter becomes active, a search is made along the genome for a pattern of bits that matches the promoter bit string.[7] If a matching region is found, the promoter is said to have *bound* to that region, and translation of the genome begins from the first bit to the right of the binding region. If no binding site is found for the active promoter, or when the translation of the current section of genome is terminated (e.g. when the Read position reaches the end of the genome, when it reaches a repressed region, or when a `stop` instruction is encountered), the active promoter is deactivated and placed at the bottom of the list in the Promoter Store, and the promoter which is now at the top of the list becomes active.

4.3.2 Repressors and the Repressor Store

The Repressor Store is a list of repressors, but, unlike in the Promoter Store, any or all of the repressors on the list may potentially be active at the same time. When a new repressor is added to the store, a search is made for a binding site on the genome,[8] in a similar way as for the active promoter. If a binding site is found, the repressor is said to be bound to the corresponding area of the genome, and that area of the genome is said to be repressed. If, during translation of the genome, the read position moves onto a repressed site, translation ceases at that point and the current promoter is deactivated.

4.4 The Translator

The process of translating the genome into executable instructions is illustrated in Figure 2. As the read head moves along the genome, it passes the string of bits that it reads to the Translator. The Translator has a table that maps bit strings to instructions in the programming language of the cells. As soon as the incoming string of bits matches an entry in this table, the Translator executes the associated instruction and the read head is moved along the genome to the next unread bit. In the current implementation, the map of bit strings to instructions is hard-coded

[5]See Section B for an explanation of the instruction set.

[6]Or on eligible InfoStrings in the Received Message Store. See Section 4.7 for details.

[7]The search begins at the current Read position on the genome, and proceeds outwards in both directions simultaneously.

[8]Or on eligible InfoStrings in the Received Message Store. See Section 4.7 for details.

into the Translator, all instructions are encoded by bit strings of equal length (six bits), and all 64 possible six-bit codes have an entry in the table (which means that in some cases, two different six-bit codes encode the same instruction). Any binary string of length six is therefore guaranteed to decode to a valid instruction. This hard-coded mapping is defined in the system input file `genetic_code.ini`, described in Section E.1.

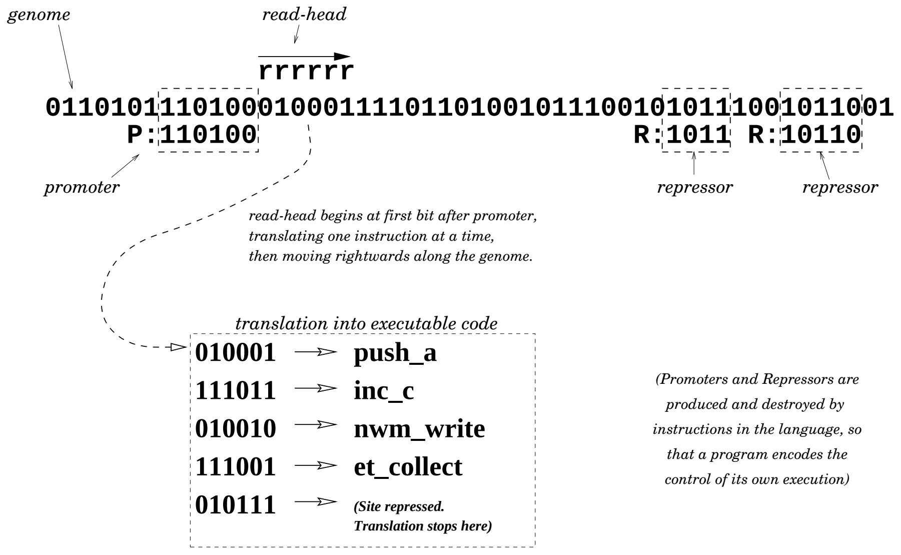

genome
read-head
rrrrrr
0110101110100010001111011010010111001010111001011001
P:110100
R:1011
R:10110
promoter
repressor
repressor
read-head begins at first bit after promoter, translating one instruction at a time, then moving rightwards along the genome.
translation into executable code
010001 → push_a
111011 → inc_c
010010 → nwm_write
111001 → et_collect
010111 → (Site repressed. Translation stops here)
(Promoters and Repressors are produced and destroyed by instructions in the language, so that a program encodes the control of its own execution)

Figure 2: Translation of the Genome.

In future experiments with the system, the hard-coded mapping from bit strings to program instructions may be replaced by a mapping which can vary from one cell to the next, and which can evolve.

4.5 The Energy Token Store

A large number of cells may exist concurrently within Cosmos. In order to run the code of all of these cells, the processor must time slice between each cell, as described in Section 3.3. In that section a formula was given which shows how many instructions a cell with a genome of a given length is allowed to execute at each time slice. However, for the cell to actually execute this number of instructions, it must pay one *energy token* to the processor for each instruction it executes. A cell has a store of energy tokens (which it collects from the environment as described in Sections 6.2 and 6.3). Furthermore, a cell's Energy Token Store may be leaky, in which case a number of energy tokens are lost from the store at the end of each time slice, in addition to any that were used to pay for the execution of instructions. The leak rate of the store is determined by the parameter `ets_leak_rate_per_timeslice`, described in Section A.

4.5.1 Cell Death

If the number of tokens in this store falls below a particular threshold (defined by the global parameter `ets_lower_threshold`, described in Section 9), the cell dies. Additionally, when the maximum number of cells allowed in the system (as defined by the parameter `max_cells_per_process`) has been reached, the processor will kill off a number of cells which have the smallest number of stored energy tokens,[9] in order to make room for new cells. It is therefore essential that a cell maintains a reasonable level of energy tokens in its store. (There is one other way in which a cell may die—it can terminate itself by executing the `kill` instruction.)

When a cell dies, any energy tokens remaining in its Energy Token Store are distributed to the local environment. More information about energy tokens is given in Section 6.2.

4.6 Cell Division and Reproduction

It has already been mentioned that a cell only has read, write and execute permission within its own boundaries. Considering that the primary function of the cells is to make copies of themselves in other areas of the system's memory, this may seem like an odd restriction. However, the mechanism of cell division and reproduction employed in Cosmos was inspired (albeit fairly vaguely) by the process of cell division in biological organisms.

The Nucleus Working Memory. Each cell has an area called the Nucleus Working Memory, which is just a WritableInfoString. The cell can compose arbitrary bit strings in this area,[10] but in the normal operation of a self-replicating program, it would construct a copy of its genome here. Thus, rather than directly writing instructions one at a time to a new area of memory (as in Tierra, for example), a Cosmos cell copies its genetic information into its own Nucleus Working Memory. When the genome has been copied in this way, the cell may issue a `nwm_divide` or a `nwm_split` instruction. These have the effect of transferring the contents of the Nucleus Working Memory into a new cell, which will be placed at a nearby grid position. The former instruction creates a cell which is completely separated from the parent cell (i.e. a new child organism), whereas the latter creates a cell which will remain a member of the same organism (i.e. an extra process in a parallel program: see Section 4.9).

In either case, upon division the contents of the Energy Token Store, Promoter Store and Repressor Store are divided equally between parent and child cell. The other main structures of the new child cell (i.e. the Nucleus Working Memory, the Received Message Store and the Communications Working Memory)[11] are initially empty.

4.7 Inter-Organism Communication Structures

Two major cell structures remain to be explained; these are the Received Message Store and the inter-organism Communications Working Memory. These two structures are both concerned with communications between organisms. The former is used to store incoming messages from other organisms, and the latter is used to compose messages to be sent out to other organisms.

The communications aspect of these structures is described in more detail in Section 7.1.2, but the part they play in the functioning of the cell is explained here.

The Communications Working Memory. The Communications Working Memory, like the Nucleus Working Memory, is a WritableInfoString (with a limited maximum length) which a cell can use to compose arbitrary sequences of bits. A cell can then issue a `cwm_send` instruction to

[9]In this situation, the choice of which cells to kill is actually stochastic, with the level of a cell's Energy Token Store determining the probability of its being killed.

[10]The only restriction is that there is a maximum length to which these strings are allowed to grow, defined by the global parameter `info_string_size_limit`. This is to prevent the situation in which a program evolves which gets stuck in an infinite loop writing to the Nucleus Working Memory, eventually using up all of the memory in the system.

[11]The function of these latter two structures is explained in Section 4.7.

broadcast the contents of the Communications Working Memory into the environment (explained in Section 6.5). The Communications Working Memory does not directly affect the functioning of the cell in any other way.

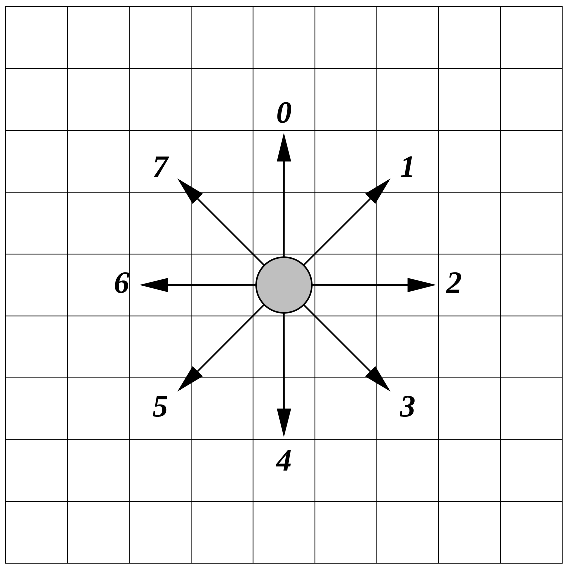
The Received Message Store. Inter-organism messages take the form of BitStrings. When they are being composed in the Communications Working Memory they are WritableInfoStrings, when they are broadcast in the environment they are converted to EnvironmentalInfoStrings, and when they are received by others cells into their Received Message Stores, they become plain InfoStrings.

A cell can issue a `rms_receive` instruction to receive messages which have been broadcast from nearby grid positions. These messages (which are EnvironmentalInfoStrings), like all InfoStrings, have a type (a number between 0 and 15) associated with them, and the value of a cell's **dx** register at the time that it issues a `rms_receive` specifies which type of messages are to be received. In addition, the search in the environment for EnvironmentalInfoStrings of the specified type only proceeds in a certain direction; starting from the grid position of the cell that issued the instruction, the search emanates in one of eight directions, specified by the low three bits of the **cx** register (see Figure 3(a)). The search proceeds one grid square at a time, covering all grid squares in the specified eighth of the area around the cell until a certain number of grid squares have been searched (defined by the global parameter `rms_receive_search_area`). For example, Figure 3(b) shows a cell searching in direction 1. If `rms_receive_search_area` is set to 12, say, then the grid positions marked with black dots will be searched. The search emanates from the cell along a series of wavefronts—the grid position on wavefront 1 is searched first, followed by those on wavefront 2, then 3, then 4. At this point, 12 positions have been visited, so the search stops. Any EnvironmentalInfoStrings of the specified type found in this area are copied into the cell's Received Message Store as InfoStrings. (A cell may extend the reach of a search by re-issuing an identical `rms_receive` instruction from the same grid position within a certain time limit after the first one. This time limit is specified by the global parameter `max_time_for_msg_receive_reinforcement`. If a cell does this, the search will continue outwards from the last grid position searched previously. In the example of Figure 3(b), the grid positions marked with gray dots, on wavefronts 5 and 6, will be the next 12 positions searched in this situation.)

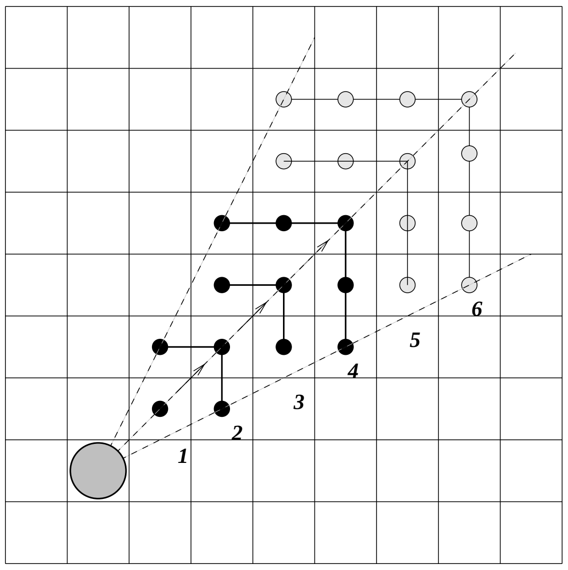
The host cell may process these received messages, using the `str_switch` and `adr` instructions to set the **ax** register to an address within a message, and using the instruction `mov_ic` to sequentially read the message.

Messages in the Received Message Store are normally treated as passive structures which may be inspected by the host code, but this is not always the case. As already mentioned, each message in the store has an associated type. The host code of the cell—the genome—being an InfoString, also has a type associated with it.[12] If any message in the Received Message Store happens to be of the same InfoString type as the cell's genome, then it may potentially be used as additional genetic material, and translated into executable instructions. In other words, promoters and repressors may bind to it in just the same way as they can bind to the genome. If the active promoter does indeed bind to a message in the Received Message Store, translation begins along it just as it would on the genome. A cell has several lines of defence against such parasitism, which are mentioned in Section 7.1.2.

[12]The type of the cell's genome cannot be directly altered, and is passed on to children when the cell splits or divides. However, it is subject to mutation like any other part of the cell (see Section 6.7). Therefore, it is possible for organisms with different genome types to emerge in the system.



(a)
A cell can search for messages in one of 8 directions



(b)
A cell searching for messages in direction 1. See text for details.

Figure 3: Searching for Communications with the `rms_receive` Instruction.

A situation where the execution of code from messages in the Received Message Store may be particularly common is when the parameter `neighbouring_genomes_readable` is set to `yes`. In this case, whenever a new promoter becomes active in the cell (see Section 4.3), rather than trying to first find a binding site on the cell's genome, or even on eligible messages already resident in the Received Message Store, the cell first imports copies of the genomes of any immediately neighbouring cells, one by one, into its Received Message Store. Note that this importation occurs automatically, without the host cell having to issue a `rms_receive` instruction, and without the neighbouring cell having to make a copy of its genome and issue a `cwm_send` instruction. Each imported message (the copy of the neighbouring cell's genome) is checked for a binding site for the new promoter. If a site is found, the message remains in the Received Message Store, and the cell starts executing instructions from it, starting in the position immediately following the binding site (see Section 4.3). If no binding site is found, the cell deletes the imported message from its Received Message Store and imports the genome of the next neighbouring cell, if there are any remaining. Only after all the neighbouring genomes have been checked in this way will the cell consider searching for a binding site on existing messages in the Received Message Store, and finally on the cell's genome itself. This mechanism was incorporated into the system in an effort to simulate the ability of programs in Tierra to read the code of neighbouring programs [Ray 91].

4.8 Other Structures

There are a number of other structures associated with a cell, which are mentioned briefly here.

Registers There are four (16 bit) registers. The registers **ax** and **bx** are used primarily for storing and manipulating addresses, whereas the registers **cx** and **dx** are used for arithmetic. The main use of the **ax** register is to store addresses returned by the `adr` instruction. This instruction looks for a specified bit string along the genome (or other eligible InfoString), and, if found, returns the address of the first bit of the matching area into the **ax** register. The address is simply the (zero-based) position of the bit from the left of the genome. The `mov_ic` instruction can be used in conjunction with `adr` to read an instruction from the genome, at the address pointed to by the **ax** register, into the **cx** register. Details of these instructions are given in Section B. (There is actually a slight complication involved with the use of `adr` and `mov_ic`; these instructions do not only work with the genome, but can also be used on InfoStrings in the Received Message Store, as already mentioned. Each cell actually keeps a pointer called the ADRStringPointer, which normally points to the genome. However, it can be changed to point to one of the InfoStrings in the Received Message Store by the use of the `str_switch` (or similar) instruction. The `adr` and `mov_ic` instructions always work on the InfoString currently pointed to by the ADRStringPointer.)

Flag There is one flag, used mainly to signal unusual or error conditions in the execution of some instructions.

Stack Each cell has a single stack, with a limited maximum capacity (defined by the global parameter `stack_size_limit`). Instructions are included in the language for pushing numbers onto the stack and for popping numbers from it.

Flaw Rate Each cell has a parameter which defines the frequency with which flaws occur in the execution of instructions (see Section 6.7). This flaw rate is subject to mutations (Section 6.7), so it may evolve over time.

Statistics and Housekeeping Information There are various other minor structures associated with a cell, mostly concerned with keeping statistics of the cell's lineage and activity (for future analysis) and with keeping track of various activities within the cell. These structures are not explained in detail here, but some are mentioned in passing throughout the rest of this chapter where appropriate.

4.9 Parallel Programs (Multicellular Organisms)

It has already been mentioned that the design of Cosmos was guided by an analogy to cellular biological organisms (Section 2). In order to model not just unicellular organisms, but also multicellular ones, Cosmos has been designed to support parallel programs—an analogy to multicellularity. Furthermore, it allows programs to dynamically create new parallel processes as they are running, as an analogy to the growth of a multicellular organism from a single celled origin.

All programs in Cosmos are instances of the Organism[13] class. An Organism may contain one or more Cells (each Cell being essentially an individual process). There is therefore no fundamental difference in the representation of serial and parallel programs; a serial program is just an Organism which has only one Cell, while a parallel program is an Organism with more than one Cell.

[13] A capital 'O' is used here to emphasise that we are talking about the specific implementation details. However, as the Organism class encapsulates the functionality of an organism, the two terms can be used interchangeably. Therefore, in the rest of the document I shall just use the term organism (with a small 'o'). The same applies for cells and the Cell class.

4.9.1 Topology of a Multicellular Organism

In a parallel program, each cell has a specific position in the environment (just like any other cell). The only restrictions on the placement of cells within a parallel program (beyond those defined for all cells by the global parameters) are that every cell within the organism must be adjacent to (i.e. occupy one of the eight neighbouring grid positions) at least one other cell owned by the organism, and that two cells within the same organism cannot share the same grid position. The topology of an organism is important in terms of its intercellular communications, as any given cell can only exchange regulators and energy tokens with immediately adjacent cells within the organism. By means of this transfer between cells in a multicellular organism, the behaviour of any cell is affected by the behaviour of its neighbours. See Section 7.1.1 for more details.

As a parallel program develops, an individual cell can actually change its position relative to its neighbours, using the `migrate` instruction. This gives a cell the opportunity of interacting with different neighbouring cells throughout the life of the program.

4.9.2 Energy Transport

As mentioned above, a cell in a multicellular organism can pass energy tokens from its store to its neighbouring cells, using the `et_transport` instruction. In this way, it is possible for a multicellular organism to develop specialised cells that collect energy tokens from the environment and distribute them throughout the rest of the organism, leaving other cells free to specialise in other tasks if necessary.

4.9.3 Fission

It has already been said that all of the cells comprising a multicellular organism are restricted to being located in such a position that they are in contact with (i.e. in an adjacent grid position to) at least one other cell in the organism. However, as individual cells within a multicellular organism can die at different times (in the ways described in Section 4.5.1), it is possible to get a situation where a collection of cells that was once connected as a multicellular organism breaks into two or more unconnected groups of cells because of the death of one of more cells in the middle of the structure (see Figure 4). If such a situation arises, the separate sub-groups of cells each now become separate organisms in their own right. Cell division and organism fission are therefore two distinct ways in which a new organism may be created.

4.9.4 The Cost of Multicellularity

The cost of being part of a multicellular organism is governed by the global parameter `multicellularity_penalty_factor`. That is, for each cell in a multicellular organism, this parameter represents the number of energy tokens that are deducted from that cell's Energy Token Store at each time slice for each additional cell with which it is in contact. For example, if a cell is adjacent to two other cells belonging to the same organism (i.e. there are two cells with which it can exchange regulators and energy tokens), then at each time slice, twice the amount of energy tokens as specified by `multicellularity_penalty_factor` are deducted from that cell's store. This parameter therefore defines how expensive it is for a cell to maintain a connection with one other cell in a multicellular organism.

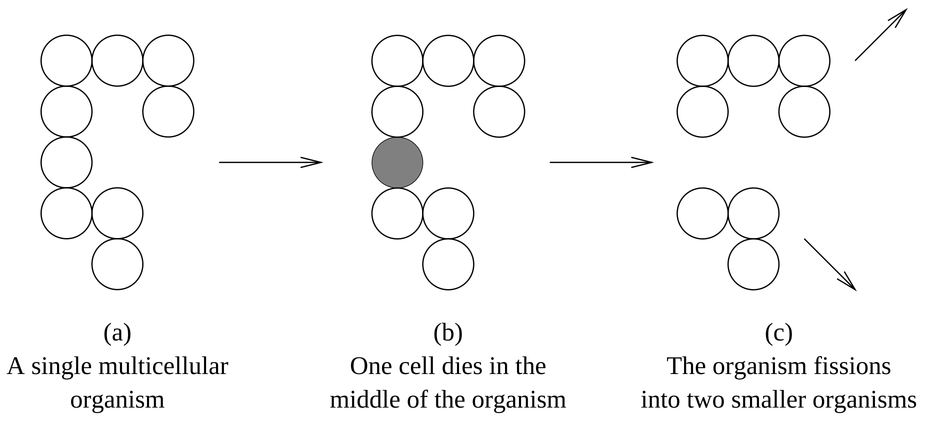

(a)
A single multicellular organism
(b)
One cell dies in the middle of the organism
(c)
The organism fissions into two smaller organisms

Figure 4: An Example of Organism Fission.

4.9.5 Organism Death

An Organism is composed of one or more cells. In Cosmos there is no specific idea of an organism, as a whole, dying—rather, an organism dies when the last of its constituent cells dies.

5 The Programming Language and Representation

REPLiCa,[14] the programming language in which the self-replicating programs are written, is based upon the Tierran language [Ray 91], with some changes and additions to support the extra functionality of Cosmos. Like Tierran, REPLiCa has been designed to be robust, in the sense that there is little syntactical structure to a program, so that any random collection of REPLiCa instructions will form a valid program that will do *something* (maybe not anything sensible, but it will not cause the system to crash). The REPLiCa instruction set is listed, with annotations, in Section B.

One big difference between Tierran and REPLiCa is in the mechanism for control flow branching and jumping. Tierran uses a system of template-driven jumping (see [Ray 91] for details). REPLiCa does not have jumps of this kind; rather, jumps may be accomplished in two different ways. The first, primarily for single jumps rather than loops, is just by the creation of an appropriate promoter to bind to the desired jump destination, either followed by the deletion from the Promoter Store of the currently active promoter (using the `reg_destroy` instruction), or by the issuing of a `stop` instruction—both of which have the effect of stopping the execution of the current section of code and activating the new promoter.[15] The second way by which (local) jumps may be performed is by the use of the `set_jmp` and `jmp` instructions. Each cell contains a pointer called the LocalJumpPointer which, if set, points to a position on the genome (or currently active InfoString in the Received Message Store). When a `set_jmp` instruction is executed, this pointer is set to the address of the next instruction. When a `jmp` instruction is executed, control passes to the instruction pointed to by the LocalJumpPointer (if it is set, otherwise no jump is performed). The LocalJumpPointer can be cleared with the `clr_jmp` instruction.

[14]'REPLiCa' is an acronym for Robust Evolvable Programming Language for Cosmos.

[15]Of course, when programs are evolving, especially when we are considering parallel programs, there may be more than one promoter in the Promoter Store at one time. However, here we are describing how a human might design a program that performs a jump—evolution would probably go about designing a program in a very different way.

The translation of the bit-string representation of a program on the genome, and the control of execution of the program by promoters and repressors, illustrated in Figure 2, has already been explained in Sections 4.2–4.4.

6 The Environment

6.1 The Grid

As mentioned in Section 3.2, cells in Cosmos live in a discrete two-dimensional spatial environment (the 'grid'). At the start of each time slice, a number of energy tokens are deposited to each position on the grid (see Section 6.2). Cells can collect these energy tokens by using the `et_collect` instruction (see Section 6.3). If energy tokens are scarce at a cell's current location (or indeed for any other reason), the cell (to be precise, the whole organism) may move around the grid (see Section 6.4). For multicellular organisms, each cell must occupy a different grid position, i.e. all organisms are 'flat' (cells cannot pile on top of each other in the same grid position). However, cells from different organisms *can* occupy the same grid position. What this means is that all organisms are flat, but they can 'slide over' each other, and in this sense the environment is two-and-a-half dimensional.

6.2 Distribution of Energy Tokens

At the start of each time slice sweep across all of the cells in the population (in the routine `DistributeEnergyTokens`, described in Section 8), the Cosmos operating system releases a certain number of energy tokens into the environment. These tokens are then available to be collected by cells, by the use of the `et_collect` instruction. At the end of each time slice sweep (in the routine `AttenuateEnvironmentalEnergy`, also described in Section 8), the operating system takes a number of energy tokens away from each grid position. In the current implementation, different grid positions may receive different numbers of energy tokens at the beginning of each time slice sweep (determined by the various distribution schemes described below), but all positions have the same number of energy tokens removed at the end of each time slice sweep (specified by the parameter `number_of_energy_tokens_per_grid_pos_per_sweep`, if they have that number available). If the number of energy tokens received by a grid position in a time slice sweep exceeds the number removed from it, and they are not collected by cells during that sweep, the excess tokens remain there for future collection. A grid position may therefore sometimes accumulate a relatively large number of energy tokens (up to a maximum limit defined by the global parameter `max_energy_tokens_per_grid_pos`) if there is not much demand for them by cells in the locality.

The distribution of energy tokens across the grid may follow a number of different patterns, defined by the global parameter `energy_distribution_scheme`. At present, four such patterns are defined: `land`, `sea`, `mixed` and `random`. Note that the total number of energy tokens distributed to the environment at each time slice sweep is always specified by the product of the parameter `number_of_energy_tokens_per_grid_pos_per_sweep` with the number of squares in the grid. The different distribution schemes determine how many of these tokens are distributed to individual squares. The different schemes work as follows:

Land Each grid position receives a constant number of energy tokens from one time slice to the next. In the current implementation, there is one extra parameter, `x_delta`, associated with this sort of energy distribution, which defines the gradient of the distribution from the left-hand side of the grid to the right-hand side. See Figures 5(a) and 5(b) for examples of this type of distribution.

Sea In contrast to `land` distribution, for `sea` distribution each grid position receives a varying number of energy tokens from one time slice to the next. During each time slice, energy tokens are distributed to grid positions which are located under a 'wave'—a vertical band which moves one position to the right after each time slice: see Figure 5(c). Grid positions which are not located under a wave in the current time slice receive no energy tokens for that time slice. In the present implementation there are two parameters associated with this method; `wave_width` and `number_of_waves`. The former specifies the width, in grid positions, of a single wave, and the latter specifies how many waves are to be fitted in to the grid from left to right (the waves are evenly spaced across the grid).

Mixed This is a mixture of `land` and `sea` distributions, with the top portion of the grid receiving energy according to the `land` distribution, and the bottom portion according to the `sea` distribution. The relative sizes of these top and bottom portions of the grid are determined by the global parameter `land_fraction`. An example is shown in Figure 5(d).

Random Energy tokens are distributed in packets with size determined by the global parameter `energy_distribution_random_chunk_size` to randomly chosen grid positions, until the correct total number of energy tokens have been distributed. An example is shown in Figure 5(e).

A multicellular organism may also pass energy tokens between its cells (using the `et_transfer` instruction), leading to the possibility of some of the cells specialising in energy token collection and distribution of these tokens to the other cells in the organism.

With such a system of CPU-time allocation, programs may potentially evolve which operate on a wide variety of time-scales. For example, very short programs may exist which quickly grab just enough energy tokens to make a copy of themselves, while much more complicated programs may coexist which gather large numbers of tokens over long periods of time, and reproduce at a much slower rate.

When a cell dies, any unused energy tokens are passed back to the local environment (where they may be collected by other organisms). This mechanism provides potential selection pressure for the evolution of organisms that kill other organisms in order to collect the energy tokens thus released into the environment. This could happen if, for example, an organism transmitted EnvironmentalInfoStrings containing the `kill` instruction, which another organism subsequently received and executed (see Sections 4.7 and 7.1 for further details of how this would work).

6.3 Collection of Energy Tokens

In the present implementation a choice of two energy collection schemes, `shared` and `private`, is provided. The global parameter `energy_collection_scheme` determines which scheme will be used.

Shared Energy. Under this scheme, when a cell issues an `et_collect` instruction to collect energy tokens from the environment, it first tries to collect spare tokens from its current grid position. However, if the grid position does not contain sufficient energy tokens, the cell then looks for other cells at the same grid position or in one of the eight neighbouring grid positions. If other cells exist in one of these nine locations, energy tokens will be extracted from the

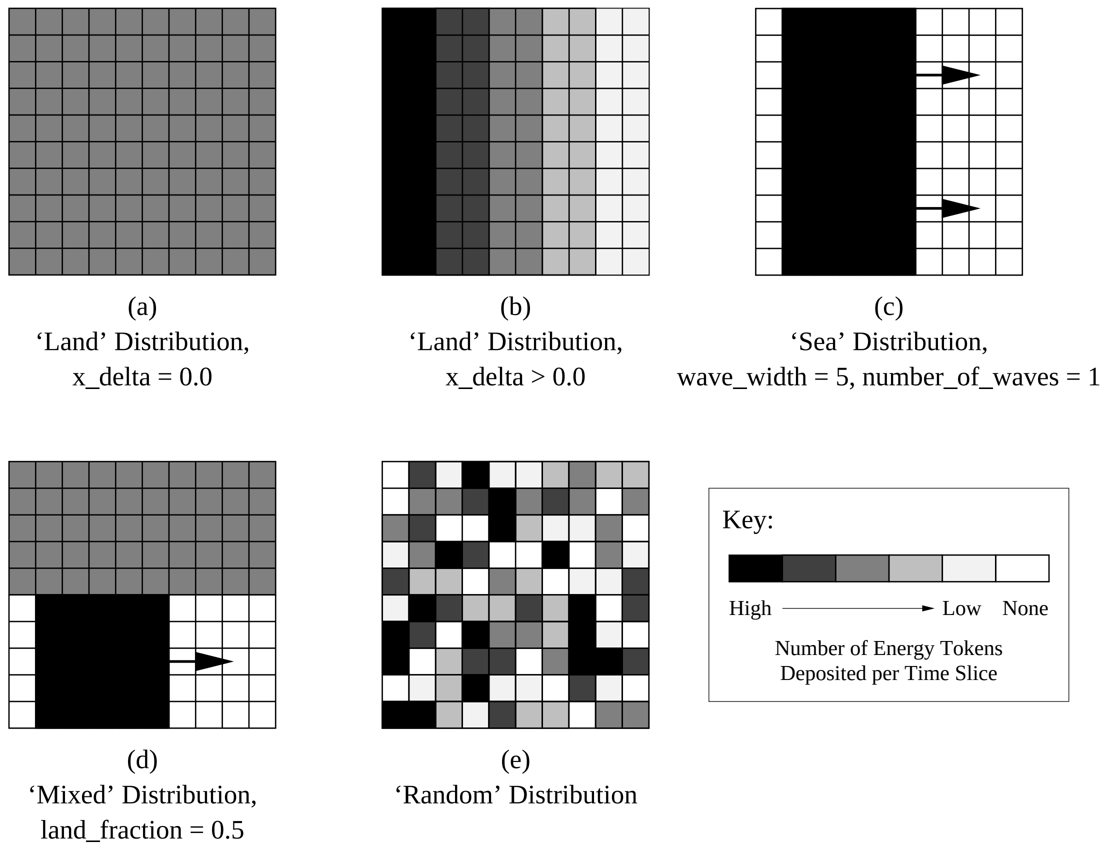


Figure 5: Different Patterns of Energy Token Distribution.

Energy Token Store of one (or more) of these (at random) until the cell has obtained the normal quota of energy tokens for one execution of `et_collect` (as defined by the global parameter `number_of_energy_tokens_per_collect`).

Private Energy. With this scheme, if the local grid position does not contain enough energy tokens for an `et_collect`, the cell just takes what is there, but does not attempt to gain additional energy tokens from neighbouring cells.

6.4 Moving around the Grid

A cell does not have to remain in its original grid position, but can move around by using the `move` instruction. The contents of the **cx** register at the time the instruction is issued determines in which direction the cell will try to move (the low 3 bits specify a direction from 0 to 7, as indicated in Figure 3(a)).

However, movement is complicated by the fact that a cell may be part of a multicellular organism (in which other cells are also trying to move, possibly in different directions). The organism must move as a whole, so what actually happens is that the issuing of a `move` instruction by a cell is actually a *vote* to move in a particular direction rather than an instruction that has immediate effect. At each time slice, an organism counts up all of the movement votes from its constituent cells, and decides how to move as follows:

A normalised total movement vector **A** is calculated by summing all the individual votes of cells within the organism:

$$\mathbf{A} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{a}_i \tag{1}$$

where n is the total number of cells in the organism, and $\mathbf{a}_i$ is the unit vector of movement (in one of eight possible directions) specified by cell i (or $\mathbf{0}$ if the cell did not issue a `move` instruction during the current time slice).

A 'multiple movement factor', M, is then calculated. This factor determines the extent to which two or more cells moving in tandem within an organism are more efficient than would be expected by simply summing their individual movements. M is defined as:

$$M = (m-1)L + 1 \tag{2}$$

where m is the number of cells within the organism that individually issued a `move` instruction; and L is the 'constant of leverage' when two or more cells move at the same time ($L \geq 0$). L is defined by the global parameter `movement_leverage_factor`.

Movement is further complicated in the situation where the organism is overlapping (or partially overlapping) another organism on the grid. In this case, there is a 'friction' term F which slows the organism down as it attempts to move over other cells. This term is defined as follows:

$$F = \frac{o}{n} \tag{3}$$

where o is the number of cells in the organism which share a grid position with cell(s) from other organisms. The friction factor can actually be turned on or off with the global parameter `apply_friction_factor`. If it is turned off, F is effectively set to zero.

The total movement that the organism attempts to make, **X**, is therefore specified by

$$\mathbf{X} = M(1-F)\mathbf{A} \tag{4}$$

The organism moves from its current position by the distance and direction given by **X**, unless it reaches the edge of the grid, in which case it stops at that point (if the grid boundary does not wrap).

6.5 Inter-Organism Communications

If a cell broadcasts an inter-organism communication using the `cwm_send` instruction (as mentioned in Section 4.7), the contents of its Communications Working Memory is packaged into an EnvironmentalInfoString structure (with an initial intensity specified by the global parameter `envinfostring_initial_intensity`, and a type specified by the low four bits of the **dx** register). This EnvironmentalInfoString is deposited in the environment in the same grid position as the cell, where it can be detected by other cells (by using the `rms_receive` instruction, described in Sections 4.7 and 7.1.2).

Each grid position in the environment can hold one EnvironmentalInfoString of each of the 16 possible types. If a string of the same type already exists in the grid position when a `cwm_send` message is issued, the existing string is deleted and replaced by the new one.

At each time slice sweep (in the `AttenuateMessageIntensities` routine, described in Section 8), the intensity of each EnvironmentalInfoString is attenuated according to the following equation:

$$I_{n+1} = k(I_n)^p \tag{5}$$

where I_n is the intensity at time n, and k and p are constants defined by the global parameters `envinfostring_decay_constant` and `envinfostring_decay_power` respectively. When the intensity of any string falls below a certain threshold (defined by the global parameter `envinfostring_lower_threshold`), the string is deleted.

There is one additional feature associated with these EnvironmentalInfoStrings, whereby a cell can reinforce the intensity of a message that it has already sent. If the cell re-issues the `cwm_send` instruction within a given number of time slices (determined by the parameter `max_time_for_msg_send_reinforcement`), while still in the same grid position, and it has not written anything else into its Communications Working Memory in the meantime, then the intensity of the existing EnvironmentalInfoString is incremented by a small amount.[16]

6.6 Environmental Information

As well as carrying specific inter-organism communications (mentioned in Section 4.7 and explained in more detail in Sections 6.5 and 7.1.2), the environment also carries summary information about itself. These messages are transmitted (in the form of EnvironmentalInfoStrings) by the environment itself, one at each grid position, and may be intercepted by cells in exactly the same way as they intercept other inter-organism communications. The messages contain the following information (represented in a binary encoding):

- The number of cells at that grid position
- The total number of free energy tokens at that grid position

All of these messages behave just like any other EnvironmentalInfoStrings in the environment; the only distinguishing feature is that they are all given an InfoString type of 15. (There are no restrictions about organisms using the same type number for their own communications.) They may be picked up by any cell using the `rms_receive` instruction.

[16]To be precise, the magnitude of the increment is kI^p, where I is the current intensity, and k and p are constants defined by the global parameters `envinfostring_decay_constant` and `envinfostring_decay_power` respectively.

6.7 Mutations and Flaws

Little has so far been said about the role of mutation in Cosmos. Mutation is a vital process from the evolutionary point of view, as it provides a continual source of genetic novelty for selection to work upon. Mutations occur naturally throughout the system at a low rate, and may affect most of the structures within the cell (i.e. the Genome, the Received Message Store, the Nucleus Working Memory, the Communications Working Memory, the Promoter Store, the Repressor Store, the flaw rate, the stack, the registers and the flag). For structures which are based upon BitStrings, mutations are governed by the global parameter `mutation_period`, which specifies the probability of an individual bit within the structure being flipped. For structures based upon integer numbers (the flaw rate, stack and registers), mutations occur at the same rate as for BitStrings, but the details are slightly different. For the flaw rate, a mutation causes a random increment or decrement in the current value within predefined limits.[17] For the stack, a mutation will, with equal likelihood, either cause a random number to be pushed onto the stack, or the top number to be popped off it. For registers, a mutation will cause the register's current value to be replaced by a random value. Mutations also affect the cell's flag at the same rate, causing the flag's state to be inverted.

In addition, variety may also be introduced into an organism by the flawed execution of instructions in its genome.[18] When a flaw occurs (which happens at a rate defined by an individual cell's flaw rate, as described in Section 4.8), the instruction which is about to be executed, rather than just being executed once, will either be executed twice (successively) or not at all. (The choice is random, with both events occurring with equal likelihood.) The effect of a flaw is therefore that instructions may occasionally produce abnormal results, such as an `inc_a` instruction adding 2 to the value of the **ax** register instead of 1.

Despite this distinction between mutations and flaws, the net results are the same. If the error affects what gets written to the Nucleus Working Memory of a cell just before it issues a `nwm_divide` instruction, then it will be passed on to the child organism and become a permanent addition to the gene pool. On the other hand, if the error does not affect the contents of the Nucleus Working Memory (even indirectly), and it does not affect the regulators that get passed on to any offspring, then it will only affect the current organism and will not be inherited by child organisms. From an evolutionary point of view, only the former scenario is important.

7 Actions and Interactions

The methods available to cells and organisms for interacting with the 'physical' environment and with other cells and organisms have already been discussed: issues such as the collection of energy tokens from the environment, and moving around the grid, were explained in Section 6; intercellular communications (i.e. the transfer of energy tokens and regulators) have been mentioned in Sections 4.9 and 6.2; and inter-organism communications have been mentioned (from the point of view of the mechanisms involved) in Sections 4.7 and 6.5. In the present section, more will be said about some higher-level effects and implications of both types of communication.

[17]To be precise, the flaw rate can change by plus or minus n parts per thousand, where n is determined by the parameter `flaw_period_max_change_per_thou`.

[18]Tierra features both mutations and flaws (although the mechanisms for flaws is somewhat different) but in subsequent work by Chris Adami and Titus Brown with their Avida system the authors suggested that flaws played only a minor role in evolution compared to mutations [Adami & Brown 94]. Informal observations from preliminary runs of Cosmos suggested that flaws in the execution of instructions significantly increase the rate at which useful mutants are produced.

7.1 Implications of Intercellular and Inter-Organism Communications

The general philosophy governing the design of the communication facilities in Cosmos was to provide the organisms with as rich an environment as possible. In particular, the inter-organism communications instructions allow organisms to exchange arbitrary messages. The idea is that, as in nature, many possibilities for communication are provided by the 'physics' of the system. The question of whether these possibilities are realised or not is left to the evolutionary process.

7.1.1 Intercellular Communications

As mentioned in Section 4.9, a cell which is a member of a multicellular organism can communicate with other cells in the organism by sending regulators from its Promoter Store and Repressor Store (using the `reg_transport` instruction). In this way, the execution of code in a particular cell may be influenced by many other cells in the organism, because regulators which are sent from one cell to another will influence which sections of code get executed in both cells. Therefore, although each cell in a multicellular organism has the same genome (assuming there are no somatic mutations), each cell may be executing different parts of this genome at any given time.

As a cell within a multicellular organism can only exchange regulators and energy tokens with its immediate neighbours, organisms adopting different shapes will have different capacities for internal communication and regulation. Within an organism, cells can also actively switch neighbourhoods by migrating to a different position (using the `migrate` instruction). If multicellular organisms do evolve in any runs of Cosmos it will be of interest to see what sorts of shapes they adopt, and how much variety in shape exists across the population.

7.1.2 Inter-Organism Communications

The mechanisms for inter-organism communications were introduced in Sections 4.7 and 6.5. A cell can broadcast an arbitrary message using the `cwm_send` instruction, and receive other messages from the environment—sent from cells in other organisms, cells within the same organism, or from the environment itself (Section 6.6)—using the `rms_receive` instruction. Allowing organisms to exchange arbitrary bit strings has little direct biological analogy. Rather, it is an attempt to equip the organisms with some communication channels in much the way that biological organisms can communicate using channels such as light, sound etc.

Once messages (InfoStrings) have arrived in a cell's Received Message Store, they may be read by the host code (using `str_switch`, `adr`, `mov_ic` and related instructions), and messages of the same type as the genome of the host cell may even be treated as executable code, as described in Section 4.7. This allows for genetic information to be exchanged between organisms in a manner analogous to the direct exchange mechanisms employed by lower biological organisms such as viruses and bacteria.

If the foreign code is detrimental to the performance of the host cell, the host may be expected to evolve measures to prevent the foreign code from being executed. This can be achieved in a number of different ways, such as by using a different type number for its own genome (which may come about by mutation), by removing the foreign code from the Received Message Store (using the `str_remove` instruction), or by not receiving the foreign code in the first place. If, however, the foreign code is beneficial to the host, then it may be expected that the host will evolve to copy this code into its Nucleus Working Memory so that it will become incorporated into the host genome in future generations. The system is even flexible enough to allow for the possibility of the evolution of sexual reproduction.

```
Inoculate
currentTimeSliceSweep = 1
while (stopping criteria not met)
{
      DistributeEnergyTokens
      AttenuateMessageIntensities
      ExecuteCellTimeSlices
      PerformOrganismLevelOperations
      if ((currentTimeSliceSweep MOD mutation_application_period) = 0)
           ApplyMutations
      if ((currentTimeSliceSweep MOD overcrowding_check_period) = 0)
           CheckOvercrowding
      if ((currentTimeSliceSweep MOD env_info_broadcast_period) = 0)
           BroadcastEnvironmentalInfo
      ExportData
      AttenuateEnvironmentalEnergy
      currentTimeSliceSweep = currentTimeSliceSweep + 1
}
```

Figure 6: The Top-Level Algorithm.

8 The Top-Level Algorithm

A pseudo-code listing of the top-level algorithm is shown in Figure 6. Most of it should be self-explanatory. The `Inoculate` routine constructs a number of self-replicating programs and places them at specified positions on the grid (governed by the parameters `ancestor`, `number` and `placement`). The stopping criteria for the main loop may be to run for a given number of time slices (if the parameter `limited_run` is set to `yes`) or to run indefinitely (only stopping if and when all programs on the grid have died out). `DistributeEnergyTokens` places a number of energy tokens in each grid position, as described in Section 6.2. `AttenuateMessageIntensities` refers to the intensities of any EnvironmentalInfoStrings that currently exist in the environment. `PerformOrganismLevelOperations` checks, for each organism, whether a fission has occurred by the death of one of more cells within it (see under "Fission" in Section 4.9), subtracts energy tokens for each cell in a multicellular organism depending on how many neighbours the cell has (see under "The Cost of Multicellularity" in Section 4.9), and finally calculates and performs any movement of the organism from the contributions made by individual cells (Section 6.4). `CheckOvercrowding` checks whether the current population of cells on the grid exceeds the limit specified by the global parameter `max_cells_per_process`. If so, a fraction of the population (specified by the parameter `population_cutback_on_overcrowding`) is killed off. The choice of which cells to kill in this situation is stochastic, but is based upon how much energy each cell has stored in its Energy Token Store. `BroadcastEnvironmentalInfo` generates an environmental message of each grid position, as described in Section 6.6. `AttenuateEnvironmentalEnergy` removes a number of energy tokens from each grid position, as described in Section 6.2.

9 Global Parameters

The Cosmos system as described contains a considerable number of global parameters. These are listed and described in Section A. The number of parameters is much larger than in most other artificial life platforms, but this is largely because other platforms often have many features which are hard-coded in a fairly arbitrary way. In contrast, Cosmos was designed to allow the user a great degree of control over the system's configuration.

10 Input and Output Files

The configuration of an individual run is specified in a number of files which Cosmos reads when the run commences. These files contain details of non-default parameter settings, of the mapping between instructions in the REPLiCa programming language and the binary encoding used to represent them in a cell's genome, and of user-defined ancestor programs. Full details of these input files are given in Section E.1.

The core Cosmos system is a stand-alone application. In order to allow the analysis of an evolutionary run, a number of log files containing information about the different organisms are written by the system during the run. The files may then be used to produce graphs and statistics about the run. These output files are described in Section E.2.

11 Major Differences between Cosmos and Tierra

In this section the main areas in which Cosmos differs from Tierra are highlighted. A fuller explanation of why some of these differences were incorporated into the system can be found in [Taylor & Hallam 97]. In the following, the extension of standard Tierra to deal with parallel processes, as described in [Thearling & Ray 94] and [Thearling 94], is referred to as 'Parallel Tierra'.

Cellular Structure. An individual program (or more precisely, an individual process, which may be serial or parallel) in Cosmos has many more structures associated with it than do programs in Tierra. Tierran programs just have the list of instructions, a program pointer, registers and a stack. In contrast, Cosmos programs also have all of the structures explained in Section 4. The idea was that they should incorporate some of the features (e.g. regulators, translation machinery, and areas where new strings may be constructed) observed in cellular biological organisms. The programs must rely largely on communications to interact with the outside world, and cannot directly read the code of their neighbours.

Regulator System. The regulator systems of Cosmos (promoters and repressors: see Section 4.3) have no equivalent in Tierra. They were designed specifically to allow cells in a multicellular organism to be able to influence which sections of code were being executed in neighbouring cells, thereby promoting cell differentiation and specialisation. The design of the regulator systems was inspired by the processes of chemical signalling between cells, and the use of promoter sequences and repressors within cells, in biological organisms.

CPU-time Allocation and Energy Tokens. In Cosmos, each cell has to pay one energy token for every instruction it executes. Cells must collect these tokens from the environment, and store them in their Energy Token Store. A cell dies when the number of tokens in its Energy Token Store falls below a threshold (defined by the parameter `ets_lower_threshold`). Furthermore, if the population size exceeds a threshold (defined by the parameter `max_cells_per_process`), cells are killed off stochastically, but those with fewer energy tokens in their Energy Token Store have a greater chance of being killed. A cell can therefore exert considerable influence over its own longevity, via its success at collecting energy tokens from the environment.

In contrast, programs in Tierra have little control over their longevity. As individual Tierran programs have no notion of energy levels, a separate 'reaper queue' mechanism is employed to govern cell death. Programs can move up the queue if they cause error conditions during execution, but in general the probability of death increases with age [Ray 91]. The reaper queue therefore effectively imposes an upper limit on the lifespan of programs, whereas there is no theoretical upper limit in Cosmos.

Additionally, the energy token scheme in Cosmos introduces the idea of a competition for the available energy—an idea which is missing in Tierra. Furthermore, if the parameter `energy_collection_scheme` is set to `shared`, cells may extract energy tokens from their neighbours. In this situation, a cell is a potential energy resource for other cells, and, if environmental energy were scarce, it would become advantageous for a cell to kill its neighbours by draining their energy. If cells could defend themselves against such attacks, some sort of coevolutionary process might arise from such interactions.

Read, Write and Execute Privileges. Tierran programs only have write access within their own 'cell membrane' (apart from when they are in the process of creating a daughter cell, when they also have write access to a specific additional chunk of memory, which has been allocated by the Tierra operating system). A similar situation exists in Cosmos. However, Tierran programs have read and execute privileges for *all* areas of instruction memory, so that they can directly examine the code of other programs, and even execute this code. Cosmos cells, on the other hand, only have direct read and execute privileges within their own cell membrane, and must rely on the system's communication facilities to interact with other cells (see Section 7.1). This restriction in Cosmos is related to the guiding analogy of the biological cell, which cannot directly read the genetic code of a neighbouring cell.

Exchange of Messages and Genetic Information. The Cosmos mechanisms for the direct exchange of arbitrary messages (which may, for example, be copies of genetic information) have no parallel in Tierra. This difference is linked to the differences in read, write and execute privileges described in the previous point.

Division Process. This point is related to the previous two. As a Cosmos cell only has write access within its own cell membrane even when it is composing a copy of itself, this copy must first be composed within the parent cell (in the Nucleus Working Memory). The copy is then issued *en masse* to a new memory location.

In Tierra, a cell is first allocated a new block of memory, then writes a copy of itself into this memory, and finally 'divides', signalling that the block of memory is now a new organism in its own right.

There is not a great deal of difference between the two mechanisms, but an advantage of the Cosmos method is that it allows an organism to *reproduce* (i.e. to create a child organism) and to *grow* (i.e. create a new cell which remains a member of the multicellular organism) using exactly the same technique.

In contrast, Parallel Tierra includes a `split` instruction which adds an additional CPU to the processor structure of the program. This mechanism is natural for a parallel machine architecture with a shared program space, as used with Parallel Tierra. In Cosmos memory is not shared across cells, so that a multicellular program must actually copy itself from one cell to another in order to run in parallel. With this type of architecture, it seems preferable that the bulk of such copying work should be performed by the cells themselves rather than by the Cosmos operating system.[19]

Additionally, having very similar mechanisms for growth and reproduction of organisms is arguably more analogous to the way that multicellular biological organisms may have evolved.

Local Competition. One of the problems that has been observed with the process of evolution in Tierra is that it suffers from premature convergence due to global interactions between cells [Adami & Brown 94].

Chris Adami and Titus Brown sought to overcome this problem in their Avida system by giving each of the cells a location on a two dimensional toroidal grid. Cells can only interact with other cells occupying nearby grid positions, thereby slowing down the rate of propagation of evolutionary changes throughout the total population and promoting heterogeneity.

Cosmos addresses this problem by placing organisms on a grid (as in Avida), and by restricting cells to only be able to communicate and interact with other cells within a certain distance on the grid.[20]

Binary Representation. In Tierra, programs are directly represented as lists of instructions. In Cosmos, the program code is represented as a binary string (specifically, an InfoString), and a translation process is required to produce the executable code. One consequence of this design is the possibility of the evolution of ultra-compact programs which use the same section of bit string to encode multiple sequences of instructions in different reading frames (as is observed in some biological organisms; [Matthews 91] p.144). Another consequence is that it would be easy to modify the system in order to study the evolution of the genetic code itself (i.e. the mapping from bit strings to program instructions).

Size of Instruction Set. The REPLiCa instruction set is about twice as big as that of the Tierran language. Many of the instructions can certainly be removed without having a great impact on the things that programs can do (e.g. the self-replicator listed in Section C.1 only uses 17 different instructions). If the genetic code were allowed to evolve, then unused instructions might be expected to be removed from the code by natural selection, allowing common instructions to be represented multiple times.

Memory Model. Cosmos uses a distributed memory model of parallelism, in contrast to the shared memory model of Parallel Tierra. In other words, each cell in a multicellular organism in Cosmos has its own copy of the program code, of the other cellular structures, and of the CPU state information (registers, instruction pointer, etc.). This distributed memory model, together with the Cosmos regulator system, should promote the emergence of differentiation in parallel programs. However, little work has so far been conducted with parallel programs in Cosmos, so it is not yet known how effective this approach really is.

Memory Addressing Scheme. For reading from and writing to structures within cells, Cosmos uses a local addressing scheme for each structure (i.e. the first bit of the Genome, of the Communications Working Memory, and of the messages in the Received Message Store, are all treated as address zero within that particular structure). Cells have no knowledge of their memory location (or that of other cells) in the global addressing scheme of the system. This is in

[19]But see the further discussion on this topic in Section 7.2 (pp.208–212) of [Taylor 99].

[20]But see the further discussion on this topic in Section 7.2 (pp.215–219) of [Taylor 99].

contrast to Tierra, which uses a global addressing scheme. The only ways that cells can interact with each other are therefore by communication; by physical contact, such as by extracting energy tokens from each other (which is possible when the parameter `energy_collection_scheme` is set to `shared`—see Section 6.3) and slowing down passing organisms (see Section 6.4); and, for cells within a multicellular organism, by the exchange of regulators and energy tokens.

Appendices

A Global Parameters

An annotated list of all of the parameters available in Cosmos is presented in this section. These are grouped into a number of different categories according to their function. The user may specify non-default values for these parameters in the Cosmos input file `params.ini`, described in Section E.

A.1 Inoculation

`ancestor` (*type*: enumerated, *range*: {`a1,a2,user_defined`})
: Specifies the ancestor(s) programs to be used for inoculation. There are two predefined ancestors (`a1` and `a2`, listed in Section C). If `user_defined` is specified, the ancestor(s) are read from the file `ancestor.ini`. The format of this file is described in Section E.

`number` (*type*: non-negative integer, *range*: 1–10000)
: The number of individual programs to inoculate the system with at the start of the run. If more than one type of ancestor is specified in the `ancestor.ini` file, these are introduced alternately until a total of `number` individuals is reached. If the parameter `placement` is set to `even`, then the actual number of inoculated individuals may be slightly smaller than that specified by `number` (see description of `placement` for details).

`placement` (*type*: enumerated, *range*: {`even,random`})
: Determines the placement of the inoculated ancestors. For `even` placement, the ancestors are placed evenly on the grid in a square pattern, where the sides of the square are as close as possible to the square root of the number specified by the parameter `number`. If `number` is not a square number, the actual number of individuals will therefore be slightly less than specified. For `random` placement, individuals are placed completely randomly, and no check is made to see whether the chosen position is already occupied.

A.2 Start of Run

`rng_seed` (*type*: integer, *range*: any)
: Used to seed the pseudo-random number generator at the start of the run. If `rng_seed` is negative, then an arbitrary seed is chosen (based upon the current clock time).

`comment` (*type*: character string, *range*: any)
: An optional description of the run, which will appear in the `run.log` output file.

`restart` (*type*: boolean, *range*: {`yes,no`})
: A value of `yes` will cause an interrupted run recorded in the file specified by the parameter `restart_file` to be restarted.

`restart_file` (*type*: character string, *range*: any)
: The name of the file to be used to restart an interrupted run (see `restart`).

`run_neutral_model` (*type*: boolean, *range*: {`yes,no`})
: If set to `yes`, a neutral model is run based upon data recorded in the input file `neutral.dat`. This file is generated during a previous run in which `record_neutral_model_data` is set to `yes`. For an explanation of neutral models, see Section 5.1.4 (p.111) of [Taylor 99].

A.3 Termination

`limited_run` (*type*: boolean, *range*: {`yes`,`no`})
If `yes`, run will stop after the number of time slices specified by the parameter `number_of_timeslices`. Otherwise, the run will continue indefinitely.

`number_of_timeslices` (*type*: non-negative integer, *range*: any)
See `limited_run`.

A.4 Environment

`grid_size` (*type*: positive integer, *range*: any)
Specifies the number of squares along each direction of the grid.

`horizontal_wrap` (*type*: boolean, *range*: {`yes`,`no`})
Specifies whether the grid wraps around in the horizontal direction.

`vertical_wrap` (*type*: boolean, *range*: {`yes`,`no`})
Specifies whether the grid wraps around in the vertical direction.

`max_cells_per_process` (*type*: non-negative integer, *range*: any)
Specifies an absolute population ceiling for the number of cells in the environment.

`population_cutback_on_overcrowding` (*type*: real number, *range*: any)
If the number of cells in the environment exceeds `max_cells_per_process`, then a proportion of the population, specified by `population_cutback_on_overcrowding`, is killed off. Cells to be killed are chosen stochastically, but based upon the number of energy tokens they have stored.

`overcrowding_check_period` (*type*: positive integer, *range*: any)
Specifies the period (expressed as a number of time slice sweeps) between successive checks for population overcrowding.

`number_of_energy_tokens_per_grid_pos_per_sweep`
(*type*: non-negative integer, *range*: any)
The average number of energy tokens distributed to each grid position at the beginning of each time slice sweep. The number of tokens distributed to individual squares may vary, as determined by the parameter `energy_distribution_scheme`. This parameter also determines the number of energy tokens taken away from each grid position at the end of each time slice sweep. See Section 8.

`max_energy_tokens_per_grid_pos` (*type*: non-negative integer, *range*: any)
The maximum number of free energy tokens that any square in the environment can store. If additional tokens are deposited on a square which already contains the maximum number allowed, the extra tokens are lost.

`env_info_broadcast_period` (*type*: non-negative integer, *range*: any)
Specifies the period (expressed as a number of time slice sweeps) between broadcasts of environmental information. See Section 6.6.

`envinfostring_decay_constant` (*type*: real number, *range*: any)
Governs the decay rate of messages in the environment. See Section 6.5.

`envinfostring_decay_power` (*type*: real number, *range*: any)
Governs the decay rate of messages in the environment. See Section 6.5.

`envinfostring_lower_threshold` (*type*: real number, *range*: any)
Specifies a threshold intensity for messages in the environment, below which they are deleted. See Section 6.5.

`envinfostring_initial_intensity` (*type*: real number, *range*: any)
Specifies the intensity assigned to newly created environmental messages. See Section 6.5.

`max_time_for_msg_send_reinforcement` (*type*: non-negative integer, *range*: any)
Specifies the maximum time interval (in number of time slices) in which a cell can reinforce the intensity of a message it has previously sent using the `cwm_send` instruction. See Section 6.5.

`max_time_for_msg_receive_reinforcement` (*type*: non-negative integer, *range*: any)
Specifies the maximum time interval (in number of time slices) in which a cell can extend the search area of a previously issued `rms_receive` instruction. See Section 4.7.

`rms_receive_search_area` (*type*: non-negative integer, *range*: any)
Specifies the number of squares searched for environmental messages upon each execution of the `rms_receive` instruction. See Section 4.7.

`energy_collection_scheme` (*type*: enumerated, *range*: {`private,shared`})
Specifies the rules governing the collection of energy tokens by a cell from the environment and from neighbouring cells. See Section 6.3.

`energy_distribution_scheme` (*type*: enumerated, *range*: {`land,sea,mixed,random`}) Specifies how energy tokens are distributed across the environment by the Cosmos operating system at the beginning of each time slice sweep. See Section 6.2.

`energy_distribution_random_chunk_size` (*type*: non-negative integer, *range*: any)
Specifies how many energy tokens are distributed to each randomly chosen square when `energy_distribution_scheme` is set to `random`. See Section 6.2.

`x_delta` (*type*: real number, *range*: any)
Specifies the energy gradient when `energy_distribution_scheme` is set to `land` (or `mixed`). See Section 6.2.

`wave_width` (*type*: positive integer, *range*: any)
Specifies the width of energy wave columns (expressed in number of squares) when `energy_distribution_scheme` is set to `sea` (or `mixed`). See Section 6.2.

`number_of_waves` (*type*: positive integer, *range*: any)
Specifies the number of energy waves, each of width `wave_width`, are fitted across the grid when `energy_distribution_scheme` is set to `sea` (or `mixed`). See Section 6.2.

`land_fraction` (*type*: real number, *range*: 0.0–1.0)
Determines the proportion of the environment to be treated as `land` when the parameter `energy_distribution_scheme` is set to `mixed`. An integer number of rows to be treated as `land` is calculated by rounding down the product of `land_fraction` and `grid_size`. These `land` rows are always at the top of the grid, and the `sea` rows at the bottom.

A.5 Organism

`max_cells_per_organism` (*type*: non-negative integer, *range*: any)
  Specifies the maximum number of cells in a multicellular organism.

`movement_leverage_factor` (*type*: non-negative real number, *range*: any)
  Partially specifies how a multicellular organism moves as a result of its constituent cells trying to move. See Section 6.4.

`apply_friction_factor` (*type*: boolean, *range*: {`yes`,`no`})
  Determines how organisms move when two or more cells occupy the same square in the environment. See Section 6.4.

`multicellularity_penalty_factor` (*type*: real number, *range*: any)
  Specifies a cost for multicellularity, in the form of a number of energy tokens removed from each cell in a multicellular organism at each time slice, depending on how many other cells it neighbours within the organism. See Section 4.9.4.

A.6 Cell

`ets_lower_threshold` (*type*: non-negative integer, *range*: any)
  Specifies a threshold number of energy tokens in a cell's Energy Token Store, below which the cell dies.

`ets_leak_rate_per_timeslice` (*type*: non-negative integer, *range*: any)
  Specifies the number of energy tokens removed from each cell's Energy Token Store at each time slice, on top of those removed for executing instructions. See Section 4.5.

`et_value_constant` (*type*: real number, *range*: any)
  Partially determines the number of instructions a given cell is allowed to execute at each time slice. See Section 3.3.

`et_value_power` (*type*: real number, *range*: any)
  Partially determines the number of instructions a given cell is allowed to execute at each time slice. See Section 3.3.

`default_ets_level_of_ancestor` (*type*: non-negative integer, *range*: any)
  Specifies the default number of energy tokens given to each inoculated ancestor program at the start of the run. This default can be overridden if a different number is specified in the `ancestor.ini` file for a user-defined ancestor.

`number_of_energy_tokens_per_collect` (*type*: non-negative integer, *range*: any)
  Specifies the number of energy tokens that a cell will attempt to collect from the environment for each execution of the `et_collect` instruction. The actual number of energy tokens collected depends upon availability. See Section 6.3.

`max_energy_tokens_per_cell` (*type*: non-negative integer, *range*: any)
  Specifies the maximum number of energy tokens that a cell can store in its Energy Token Store.

`info_string_size_limit` (*type*: positive integer, *range*: any)
  Specifies the maximum length of any InfoString object in the system. This imposes an upper limit on the size of genomes, environmental messages, etc.

`stack_size_limit` (*type*: non-negative integer, *range*: any)
Specifies the capacity (maximum number of items) of the cells' stacks.

`rms_size_limit` (*type*: non-negative integer, *range*: any)
Specifies the capacity (maximum number of messages) of the cells' Received Message Stores.

`neighbouring_genomes_readable` (*type*: boolean, *range*: {`yes`,`no`})
Specifies whether the genomes of neighbouring cells are imported as messages into the Received Message Store and checked for binding sites when a newly active promoter is searching for a binding site. See Section 4.7.

A.7 Mutations and Flaws

`apply_mutations` (*type*: boolean, *range*: {`yes`,`no`})
Specifies whether mutations are to be operative during the run. If set to `no`, then the associated parameters `mutation_period` and `mutation_application_period` have no effect.

`mutation_period` (*type*: non-negative integer, *range*: any)
Specifies the expected number of bits within the cells of all the organisms in the population that will be unaffected by mutations between successive bits which *are* affected, at each application of the mutation procedure.

`mutation_application_period` (*type*: non-negative integer, *range*: any)
Specifies the period (expressed as a number of time slice sweeps) between successive applications of the mutation procedure.

`apply_flaws` (*type*: boolean, *range*: {`yes`,`no`})
Specifies whether the flawed execution of instructions is to be operative during the run. If set to `no`, then the associated parameters `default_flaw_period` and `flaw_period_max_change_per_thou` have no effect.

`default_flaw_period` (*type*: non-negative integer, *range*: any)
Specifies the default flaw period initially associated with inoculated ancestor programs. This is the expected number of successful executions of instructions by the Cosmos operating system between successive flawed executions. This default can be overridden if a different number is specified in the `ancestor.ini` file for a user-defined ancestor.

`flaw_period_max_change_per_thou` (*type*: non-negative integer, *range*: any)
Specifies the degree to which a cell's flaw period may be changed by a single mutation. Expressed in parts per thousand. The flaw rate may be mutated to any number in the range of its current value plus or minus the specified fraction of that value.

A.8 Input and Output

`species_count_export_period` (*type*: non-negative integer, *range*: any)
Specifies the period (expressed as a number of time slice sweeps) between successive outputs to the file `concentrations.dat`.

`species_count_threshold_for_recording` (*type*: non-negative integer, *range*: any)
Specifies the minimum number of individuals of a given species (genotype) that must

coexist in the population before information about that species is written to the file `species_current.dat`.

`max_output_file_size` (*type*: non-negative integer, *range*: any)
Specifies the maximum size (in number of bytes) of output files. When an output file exceeds this threshold it is closed and compressed, and a new file (with a different extension) is opened for writing. See Section E.

`morgue_record_period` (*type*: non-negative integer, *range*: any)
Specifies the expected number of deaths of eligible organisms between successive recordings of information about the death of an eligible organism (i.e. an organism of a genotype that has already been recorded in the file `species_current.dat`) into the file `morgue.dat`.

`backup_period` (*type*: non-negative integer, *range*: any)
Specifies the period (expressed as a number of time slice sweeps) between successive backups of the run being written to the file `autosave.ser`.

`record_neutral_model_data` (*type*: boolean, *range*: {`yes`,`no`})
Specifies whether data for the run is to be written to the file `neutral.dat` for subsequent playback as a neutral model. See also `run_neutral_model`.

`neutral_model_data_export_period` (*type*: non-negative integer, *range*: any)
Specifies the period (expressed as a number of time slice sweeps) between successive recordings of information to the file `neutral.dat`. Only relevant if the parameter `record_neutral_model_data` is set to `yes`.

`group_zero_length_genotypes` (*type*: boolean, *range*: {`yes`,`no`})
Specifies whether all organisms of zero length are to be regarded as belonging to the same genotype (i.e. 0AAAA) for the purposes of data collection and analysis.

`visualisation_recording_on` (*type*: boolean, *range*: {`yes`,`no`})
Specifies whether visualisation output files ('movie' files) are to be recorded for the run.

`visualisation_record_energy_only` (*type*: boolean, *range*: {`yes`,`no`})
Specifies whether only the energy-related visualisation files will be recorded, or whether they all will be. This is only relevant if `visualisation_recording_on` is set to `yes`.

`visualisation_intersample_period` (*type*: non-negative integer, *range*: any)
Specifies the period (expressed as a number of time slice sweeps) between the beginning of recording of successive samples of the visualisation data. This is only relevant if the parameter `visualisation_recording_on` is set to `yes`.

`visualisation_intrasample_period` (*type*: non-negative integer, *range*: any)
Specifies the period (expressed as a number of time slice sweeps) between successive recording of visualisation data within a single sample period. This is only relevant if the parameter `visualisation_recording_on` is set to `yes`.

`visualisation_sample_size` (*type*: non-negative integer, *range*: any)
Specifies the number of data points (i.e. the number of recorded time slices) for each sample in the visualisation data. Only relevant if the parameter `visualisation_recording_on` is set to `yes`.

B The REPLiCa Instruction Set

The instruction set of the REPLiCa programming language contains 62 instructions in total, as listed below. The user is able to include and exclude any of these instructions from the available instruction set for any particular run of the system, according to the specification of the genetic code in the input file `genetic_code.ini` (described in Section E).

In the following description of the instructions, `RMS` stands for Received Message Store, `CWM` for Communications Working Memory, and `NWM` for Nucleus Working Memory. `ADRString` refers to the string pointed to by `ADRStringPointer` (mentioned in Section 4.8); this is the InfoString upon which the `adr` and `mov_ic` instructions will act. This may be the cell's own genome, or it may be a message in the cell's Received Message Store. `ADRStringPointer` can be changed to point to a different InfoString with the `str_switch` and similar instructions. A register enclosed in square brackets (e.g. `[ax]`) indicates the contents of the memory location specified by the value of that register.

Some of the instructions relating to regulators (e.g. `reg_create`), and to searching for binding sites (e.g. `adr`), must be immediately followed by a valid binding site specification if they are to operate correctly. A valid specification is a consecutive string of `nop` instructions (i.e. taken from the set {`nop_00`, `nop_01`, `nop_10`, `nop_11`}).

A few instructions involve actions which occur in a particular direction relative to the cell executing them (e.g. `move`, `et_transport`). In these cases, the direction is specified by the low 3 bits of the `cx` register. This gives a number between 0 and 7, which corresponds to the directions shown in Figure 3(a).

- Register Manipulation Operations

```
push_a        ; push ax onto stack
push_c        ; push cx onto stack
pop_a         ; pop stack into ax
pop_c         ; pop stack into cx

swap_ab       ; ax=bx, bx=ax
swap_cd       ; cx=dx, dx=cx

mov_ic        ; copy one instruction from ADRString, starting
              ; from address [ax], into cx. The length of the
              ; instruction copied is written to dx. ax += dx.
              ; If ax>length of ADRString, flag=true.

clr_f         ; flag=false

inc_a         ; increment ax (if overflow, flag=true)
inc_c         ; increment cx (if overflow, flag=true)
dec_c         ; decrement cx (if underflow, flag=true)

add_cd        ; cx=cx+dx (if overflow, flag=true)
sub_cd        ; cx=cx-dx (if underflow, flag=true)
sub_ab        ; cx=ax-bx (if underflow, flag=true)
```

```
zero_c          ; cx=0
not_c           ; cx=NOT cx (bitwise)
and_cd          ; cx=cx AND dx (bitwise)
or_cd           ; cx=cx OR dx (bitwise)
shl_c           ; shift bits in cx left
                ;  (lo bit <- flag, hi bit -> flag)
shr_c           ; shift bits in cx right
                ;  (hi bit <- flag, lo bit -> flag)
not_lo_c        ; flip low bit of cx
```

- Flow of Control Operations

```
if_fl           ; if (flag=false) increment instruction pointer
                ; otherwise do nothing

if_not_fl       ; if (flag=true) increment instruction pointer
                ; otherwise do nothing

if_z            ; if (cx!=0) increment instruction pointer
                ; otherwise do nothing

stop            ; stop execution and unbind current promoter

set_jmp         ; point the local jump marker to the next
                ; instruction

clr_jmp         ; clear the local jump marker

jmp             ; if local jump marker is set, jump to that
                ; instruction, otherwise do nothing (set flag=true)
```

- Nucleus Working Memory

```
nwm_clear       ; Erase the NWM WritableInfoString

nwm_write       ; Copy first n bits of cx to the end of the NWM,
                ; where n is given by the low 4 bits of dx.

nwm_write_bit   ; Copy the first bit of cx to the end of the NWM.

nwm_divide      ; Create a new single-celled organism by copying
                ; NWM WritableInfoString as the new genome,
                ; splitting the contents of the regulator stores
                ; and Energy Token store, and creating an initially
                ; empty RMS and CWM. The NWM of parent cell is
                ; empty after the division. Child cell is placed
                ; randomly at a free location near the parent (no
                ; preferred direction).

nwm_split       ; Transfer contents of NWM into a new cell which
                ; will become an additional process of the
                ; multicellular organism. Child cell is placed in a
                ; position relative to the parent specified by the
                ; low 3 bits of the cx register. If this location
                ; is already occupied, the nearest free neighbour
                ; is occupied. If all 8 neighbours are occupied,
                ; child cell replaces the parent (parent dies).
```

- (Inter-organism) Communications Working Memory

```
cwm_clear       ; Erase the CWM WritableInfoString

cwm_write       ; Copy first n bits of cx to the end of the CWM,
                ; where n is given by the low 4 bits of dx.

cwm_write_bit   ; Copy first bit of cx to the end of the CWM.

cwm_send        ; Transfer contents of CWM to an
                ; EnvironmentalInfoString at the current grid
                ; position of the cell, with a type given by the
                ; low 4 bits of dx. The msg is given a standard
                ; intensity. This msg replaces any existing msg at
                ; that grid posn with the same msg type. After the
                ; instruction is issued, the CWM is emptied. If
                ; another cwm_send is sent within n time slices of
                ; the last one (and msg type is the same, and CWM
                ; is now empty), the intensity of the existing msg
                ; at that grid pos with msg type=dx is increased
                ; (by a standard amount).
```

- Received Message Store

```
rms_receive     ; Receive msg(s) from environment. One execution of
                ; this instruction will search over a catchment
                ; area of 1/8th of a full circle (45 degrees), in a
                ; direction specified by the low 3 bits of the cx
                ; register. The search is for messages of String
                ; type specified by the low 4 bits of the dx
                ; register. Each search initially spreads out from
                ; the cell and covers a fixed number (n) of grid
                ; cells (specified by the parameter
                ; rms_receive_search_area) and all msgs of the
                ; right type are received and added to the end of
                ; the RMS. If another rms_receive is issued for the
                ; same String type and same direction within a
                ; fixed number of time slices (specified by the
                ; max_time_for_msg_receive_reinforcement
                ; parameter), the current search continues outwards
                ; (covering n more grid positions).
```

- Energy token collection / transfer

```
et_collect      ; if (environmental energy token available)
                ;       pick up n tokens from environment
                ;       (n specified by global parameter
                ;       number_of_energy_tokens_per_collect)
                ; else
                ;       flag=true

et_transport    ; if ((number of tokens in store >=
                ;     ets_lower_threshold)
                ;     &&
                ;     (there is a cell belonging to the same
                ;     organism in the direction indicated by the
                ;     lower 3 bits of cx))
                ;       send n tokens to neighbouring cell in
                ;       direction indicated (n specified by
                ;       the global parameter
                ;       number_of_energy_tokens_per_collect)
                ; else
                ;       flag=true

et_check        ; cx=current level of energy token store
```

- Regulators

```
[the following instructions all work for both promoters and
repressors. To work correctly, they must both be followed by two or
more nop's. The first nop specifies whether a promoter or a
repressor is being referred to (nop_00 and nop_01 indicate a
promoter, and nop_10 and nop_11 a repressor). The second and
subsequent nop's specify the binding pattern of the regulator.]

reg_destroy     ; if ((valid binding site specification follows
                ;     instruction) &&
                ;     (a matching regulator exists in the store))
                ;       remove one of the matching regulators
                ; else
                ;       flag=true

reg_transport   ; if ((valid binding site specification follows
                ;     instruction) &&
                ;     (a matching regulator exists in the store) &&
                ;     (there is a cell belonging to the same
                ;     organism in the direction indicated by the
                ;     lower 3 bits of cx))
                ;       send one matching regulator to neighbour
                ;       cell indicated (removing it from store in
                ;       first cell).
                ; else
                ;       flag=true

reg_create      ; if two or more nop's follow, create a regulator
                ; from the specified bit pattern and place it in
                ; the appropriate regulator store. If the first nop
                ; is a nop_00 or a nop_01, create a promoter with
                ; bit pattern specified by the second and
                ; subsequent nop's, and place it at the bottom of
                ; the list in the Promoter Store. Otherwise (if the
                ; first nop is a nop_10 or a nop_11), create a
                ; repressor with bit pattern specified by the
                ; second and subsequent nop's, and place it in the
                ; Repressor Store, checking for possible binding
                ; sites on the Genome (and other eligible
                ; InfoStrings in the Received Message Store).
```

- NOPs / Binding Site Specification

```
nop_00          ; symbols for specifying binding sites (used in
nop_01          ; creating promoters and repressors, and by
nop_10          ; adr instructions)
nop_11          ;
```

- Searching for Binding Sites

```
adr             ; if ((valid binding site specification follows
                ;     instruction) &&
                ;     (a matching binding site is found on the
                ;     ADRString))
                ;       ax = address of the memory location
                ;            immediately succeeding the
                ;            nearest matching template
                ; else
                ;       flag=true

adrf            ; as adr, but only searches forwards from current
                ; position of read-head on ADRString

adrb            ; as adr, but only searches backwards from current
```

```
                 ; position of read-head on ADRString

str_switch       ; if (there exists an InfoString with type=low 4
                 ;    bits of dx)
                 ;       ADRString=first matching String found
                 ; else
                 ;       flag=true

str_switchf      ; as str_switch, but only searches forwards from
                 ; current ADRString

str_switchb      ; as str_switch, but only searches backwards from
                 ; current ADRString

str_host         ; ADRString=cell's genome String

str_latest       ; ADRString=last String in RMS list

str_next         ; ADRString=next String in RMS list. If at end,
                 ; loop back to the cell's genome String. If on the
                 ; genome String, move to first String in RMS.

str_previous     ; ADRString=Previous String (i.e. just the
                 ; reverse of the action of str_next)

str_remove       ; if (there exists an InfoString with type=low 4
                 ;     bits of dx in the RMS)
                 ;       remove first matching string found
                 ; else
                 ;       flag=true
```

- Cell Movement Operations

```
move             ; Attempt to move cell (and whole organism) in the
                 ; direction specified by the low 3 bits of the cx
                 ; register. For multicellular organisms, each cell
                 ; that issues a move instruction during a time
                 ; slice is actually casting a vote for the desired
                 ; direction of movement. The overall effect of this
                 ; is given by a formula described elsewhere.

migrate          ; Attempt to move cell relative to other cells in
                 ; the organism in a direction specified by the low
                 ; 3 bits of the cx register. If direction if full,
                 ; nearest free direction is taken. If no free
                 ; direction is available, migration has no effect
                 ; (flag=true). Note that migration for a single
                 ; celled organism has the same effect as a move
```

```
                  ; instruction.
```

- Killing the current process (cell)

```
kill            ; Kill current cell. Any energy tokens in the
                ; cell's Energy Token Store are added to the
                ; current grid position's store. Note that if cell
                ; was part of a multicellular organism, cell death
                ; may lead to the organism physically breaking up
                ; into two or more distinct organisms.
```

C Predefined Ancestor Programs

C.1 Ancestor A1

This self-reproducing program operates by copying itself one instruction at a time into the Nucleus Working Memory. It is assumed that the program starts at memory location zero (as is usually the case); no attempt is made to look for a binding pattern to calculate the start address. Similarly, copying continues until an execution of the instruction `mov_ic` sets the flag, which indicates that the end of the program has been reached. When copying is complete, the `nwm_divide` instruction is issued to produce the offspring. A promoter is provided that will attach itself to the beginning of the program to initiate execution. A new promoter of the same type must be produced by the program itself, to be passed on to its offspring. The program listing is as follows:

1-2	`101100111000`	Start marked with a specific binding pattern
3	`et_collect`	Collect some energy
4	`nwm_clear`	
5	`zero_c`	
6	`push_c`	
7	`pop_a`	ax=0 (i.e. points to start of program)
8	`set_jmp`	
9	`et_collect`	
10	`clr_f`	
11	`mov_ic`	Main loop:
12	`if_not_fl`	Copy instructions one at a time into
13	`nwm_write`	the Nucleus Working Memory, and check
14	`if_fl`	whether end of program has been reached
15	`clr_jmp`	
16	`jmp`	
17	`reg_create`	Create a new regulator:
18	`nop_00`	The regulator will be a promoter
19	`nop_10`	
20	`nop_11`	These nop's specify a promoter
21	`nop_00`	that match the binding pattern
22	`nop_11`	at the beginning of this program
23	`nop_10`	
24	`nop_00`	
25	`nwm_divide`	
promoter	`101100111000`	Initial promoter

C.2 Ancestor A2

This self-reproducing program works in a similar fashion to ancestor A1. The difference is that it explicitly searches for its beginning and end positions by looking for appropriate binding sites, rather than assuming that copying should start from memory address zero and continue until execution of the instruction `mov_ic` sets the flag. The program listing is as follows:

```
1-2  101100111000     Start marked with a specific binding pattern
  3  et_collect
  4  et_collect
  5  nwm_clear
  6  adrb           |
  7  nop_10         |
  8  nop_11         |  Search for binding pattern
  9  nop_00         |  at start of program
 10  nop_11         |
 11  nop_10         |
 12  nop_00         |
 13  push_a
 14  pop_c
 15  et_collect
 16  dec_c          |
 17  dec_c          |
 18  dec_c          |
 19  dec_c          |  We now have to subtract the
 20  dec_c          |  length of the binding pattern
 21  dec_c          |  (12 bits) to get the address
 22  dec_c          |  of the actual start of the
 23  dec_c          |  program
 24  dec_c          |
 25  dec_c          |
 26  dec_c          |
 27  dec_c          |
 28  push_c
 29  pop_a
 30  swap_ab
 31  et_collect
 32  adrf           |
 33  nop_01         |
 34  nop_00         |  Search for binding pattern
 35  nop_11         |  at end of program
 36  nop_00         |
 37  nop_01         |
 38  nop_11         |
 39  swap_ab
 40  set_jmp
 41  et_collect
 42  et_collect
 43  clr_f
 44  mov_ic
 45  push_c
 46  swap_cd
 47  push_c
 48  swap_ab
```

49	sub_ab	
50	swap_ab	
51	pop_c	
52	swap_cd	
53	pop_c	
54	if_not_fl	
55	nwm_write	
56	if_fl	
57	clr_jmp	
58	jmp	
59	et_collect	
60	reg_create	
61	nop_00	
62	nop_10	
63	nop_11	
64	nop_00	
65	nop_11	
66	nop_10	
67	nop_00	
68	nwm_divide	
69	stop	
70-71	010011000111	End marked with a specific binding pattern
promoter	101100111000	

D Running Cosmos

Cosmos is started with the following command:

```
cosmos [OutputDirectory] [InputDirectory]
```

`OutputDirectory` and `InputDirectory` are optional arguments to tell the program where to place output files and where to search for input files. If only one directory is specified, it is taken to be the output directory. The system is configured via the input files `params.ini`, `genetic_code.ini` and `ancestor.ini`, described in Section E. If either the input or output directory is unspecified when the program is started, the current working directory will be used by default.

E Format of Input and Output Files

The formats of the various input and output files are described below. With the exception of the automatically generated backup file (`autosave.ser`), all files are in ASCII format.

E.1 Input Files

When a Cosmos run commences, the program will search for the three files listed below. Cosmos will look for these files in the current working directory, unless a different input directory is specified as an argument when the program is started. The files `genetic_code.ini` and `params.ini` are always required. The file `ancestor.ini` is only required if the parameter `ancestor` is set to `user_defined`.

E.1.1 The Genetic Code (genetic_code.ini)

The mapping between the bit string representation of instructions in a cell's genome and the instructions listed in Section B is defined in the file genetic_code.ini. The format of the file is shown in Figure 7.

```
000000 instruction_1
000001 instruction_2
000010 instruction_3
:
111111 instruction_64
```

Figure 7: Format of the genetic_code.ini file.

Note that the file *must* contain a mapping for each of the 64 six-bit codons (although they do not have to be listed in any particular order). It is permitted for multiple codons to point to the same instruction; if this is the case, fewer than 64 instructions are therefore available for organisms to use.

E.1.2 Parameter specification (params.ini)

Non-default parameter values may be specified in this file. The format is shown in Figure 8. The allowable section names are: inoculation, startinfo, termination, environment, organism, cell, mutation and io. These correspond to the groupings of parameters in Section A. The parameter names and allowable values are as listed in Section A. The file params.ini may also contain blank lines, and comments (lines beginning with the % character).

```
[section_name_1]
parameter_name_1=value_1
parameter_name_2=value_2
:
parameter_name_N1=value_N1

[section_name_2]
parameter_name_1=value_1
parameter_name_2=value_2
:
parameter_name_N2=value_N2

:
[section_name_N]
parameter_name_1=value_1
parameter_name_2=value_2
:
parameter_name_N3=value_N3
```

Figure 8: Format of the params.ini file.

E.1.3 User-defined ancestor programs (`ancestor.ini`)

If the parameter `ancestor` is set to `user_defined`, Cosmos looks in the `ancestor.ini` file for a description of the ancestor(s) to be used to inoculate the environment at the beginning of the run. The format of this file is shown in Figure 9.

```
ancestor_1_description
###
ancestor_2_description
###
ancestor_N_description
```

Figure 9: Format of the `ancestor.ini` file.

Any number of different ancestors may be specified, each separated by the line `###`. If multiple ancestors are defined in this file, they are introduced alternately into the environment during inoculation, until the specified total number of organisms has been reached (see the description of the parameters `number` and `placement` in Section A). Each ancestor description is a consecutive sequence of lines, each of which may be any one of the following:

1. A blank line.
2. A comment (commencing with the `%` character).
3. An explicit bit string to be directly added to the ancestor's genome. Specified by a line consisting of a string composed of the characters `0` and `1`. Useful for specifying binding patterns.
4. An instruction (as listed in Section B). This has the effect of writing the bit string corresponding to the instruction (as defined in the file `genetic_code.ini`) to the ancestor's genome.
5. An instruction enclosed in square brackets `[]`. This has the effect of determining a sequence of `nop` instructions (i.e. taken from the set `nop_00`, `nop_01`, `nop_10`, and `nop_11`) corresponding to the bit string representation of the specified instruction. The bit string representation of this string of `nops` is then written to the ancestor's genome. Useful for writing code that will produce regulators which will bind to a particular sequence of instructions (without requiring the programmer to know the bit string representation of these instructions).
6. A promoter to be added to the ancestor's promoter store. Specified by a line beginning with `p:` followed by a string composed of the characters `0` and `1` representing the promoter's bit string.
7. A repressor to be added to the ancestor's repressor store. Specified in same way as promoter, but with line starting `r:`.
8. An initial energy level for the ancestor. Specified by a line beginning with `e:` followed by a number to represent the desired energy level.
9. An initial flaw period for the ancestor. Specified by a line beginning with `f:` followed by a number to represent the desired flaw period.

A valid ancestor description consists of at least one instruction and one promoter. If no initial energy level or flaw period are specified, the default values defined by the parameters `default_ets_level_of_ancestor` and `default_flaw_period`, respectively, are used. If multiple ancestors are defined, they are distributed alternately across the environment, as described in Section A under the description of the `placement` parameter.

E.2 Output Files

As the run proceeds, Cosmos writes data to various output files. These are stored in the current working directory unless a different output directory is specified as an argument when the program is started. As Cosmos runs can last for an indefinitely long time (if no limit is set on the length of the run by the parameters `limited_run` and `number_of_timeslices`), the output files could also potentially grow indefinitely large. In order to keep the files at a manageable size, Cosmos breaks down the files described below (except for `run.log`, `species_current.dat` and `autosave.ser`) in the following way: each filename is given an additional extension, which is initially `.AA` (e.g. `concentrations.dat.AA`). When the size of the output file exceeds a threshold (set by the parameter `max_output_file_size`), Cosmos closes the file, compresses it (using gzip), and opens a new file with an incremented extension name (i.e. the second file will have the extension `.AB`). Writing continues in this new file until that too reaches the threshold size, and the compression procedure is repeated.

E.2.1 General Information About Run (`run.log`)

The file `run.log` contains the following information about the run:

1. Cosmos version number.
2. Run comment (specified by the parameter `comment`).
3. Time run commenced.
4. A listing of the genetic code (as specified in the file `genetic_code.ini`). Includes a list of instructions that have multiple codon mappings, and a list of instructions which have no codon mappings.
5. A listing of the ancestor(s) being used. Includes the bit string representation and corresponding instructions, together with the initial promoters, repressors, energy level and flaw period, and the ancestor's ID number.
6. A full list of system parameters together with their values.
7. The number used to seed the random number generator. If the run has been re-started, the new seed is also listed, together with the time slice at which the run re-commenced.
8. The time at which the run finished, together with the time slice at which it stopped, and a comment to indicate why the run terminated.

All except the final item on the above list are written to the `run.log` file at the beginning of the run. The final item is written when the run terminates.

E.2.2 Species Concentrations (`concentrations.dat`)

At regular intervals determined by the parameter `species_count_export_period`, summary information about the species currently in the population is written to the file `concentrations.dat`. This file has a two-line header which is required by the `actiview` program.[21] Subsequent lines of the file are of the format:

```
TimeSliceNumber:  Species1-ID Species1-Number;Species2-ID Species2-Number; ...
;SpeciesN-ID SpeciesN-Number;
```

where `SpeciesN-Number` is the number of individuals of genotype `SpeciesN-ID` in the population at time slice `TimeSliceNumber`.

E.2.3 Species Details (`species_current.dat`, `species_extinct.dat`)

Whenever a new organism is born, a check is made to see whether the number of individuals of that genotype currently in the population exceeds a threshold defined by the parameter `species_count_threshold_for_recording`. If this threshold is exceeded, and if information about the species has not previously been recorded in the file `species_current.dat`, then a line is appended to the file containing information about the species. The format of the line is:

```
SpeciesID ParentSpeciesID TimeOfFirstOccurrence InitialReadingFrame Genome
```

where `ParentSpeciesID` is the ID of the species from which the present species is descended; `TimeOfFirstOccurrence` is the time slice in which the first organism of the species was born; `InitialReadingFrame` is the frame in which the genome of the first organism of the species is being translated, expressed as a number in the range 0–5 (because codons are 6 bits long) indicating an offset from the beginning of the genome; and `Genome` is a full listing of the species' genome (written as a bit string).

When a species which has been recorded in `species_current.dat` becomes extinct, the record of that species is removed from this file, and transferred to `species_extinct.dat`. When this happens, two extra fields of information are appended to the line describing the species: `TimeOfExtinction` and `FinalReadingFrame` (the reading frame in which the last organism of the species was being translated).

E.2.4 Information on Individual Organisms (`morgue.dat`)

When an individual organism dies, if it is of a species which has already been recorded in the file `species_current.dat`, then information about the organism is considered for recording in the file `morgue.dat`. To restrict the size of this file, eligible organisms only have a 1 in N chance of being recorded in it, where N is determined by the parameter `morgue_record_period`. Each line of the file is of the format:

[21] `Actiview` is a program developed by Emile Snyder and Mark Bedau at Reed College in the USA, to produce various summary statistics and graphs depicting the evolutionary activity of the run. These are described in Section 5.1 (pp.105-110) of [Taylor 99].

```
TimeSliceNumber SpeciesID-Numeric SpeciesID-Alpha TimeOfBirth LastReadingFrame
NumberFaithfulOffspring NumberUnfaithfulOfspring TimeOfFirstFaithfulOffspring
TimeOfSecondFaithfulOffspring FlawRateAtBirth MaximumCellsInOrganism
ForeignCodeExecution
```

where SpeciesID-Numeric and SpeciesID-Alpha are the numeric and alpha components of the SpeciesID of the organism (which are split to make subsequent extraction of organism genome length data easier); NumberFaithfulOffspring is the number of faithful offspring that the organism gave birth to (and NumberUnfaithfulOfspring has a similar meaning relating to unfaithful offspring); TimeOfFirstFaithfulOffspring is the time slice at which the organism gave birth to its first faithful offspring, or 0 if it did not achieve this (and TimeOfSecondFaithfulOffspring has a similar meaning relating to the second faithful offspring); MaximumCellsInOrganism is the maximum number of cells that the organism was composed of at any stage of its life; and ForeignCodeExecution is 1 if the organism ever executed any code from its Received Message Store during its lifetime, and 0 otherwise.

E.2.5 Phylogenetic Information (phylogeny.dat)

Information about the phylogeny (ancestry) of all species that arise during a run is recorded in the file phylogeny.dat. Each line of this file is of the format:

```
SpeciesID,ParentSpeciesID
```

where ParentSpeciesID (the immediate ancestor of SpeciesID) is set to 0 for the record of a species that was used to inoculate the system at the start of the run. The entire phylogeny of any species can therefore be reconstructed from the data in this file, right back to an ancestor used to inoculate the system at the start of the run; the Perl script phyl will print the full phylogenetic tree for a species passed in as an argument.

E.2.6 Neutral Model Data (neutral.dat)

If the parameter record_neutral_model_data is set to yes, Cosmos will record data about the run in the file neutral.dat. The period between successive updates to this file is set by the parameter neutral_model_data_export_period. The data in neutral.dat can subsequently be used to run a neutral shadow of the run (see the description of the parameter run_neutral_model in Section A). The first two lines of the file are a header: the first line records the size of the environment (as specified by the parameter grid_size), the number of organisms with which the system was inoculated at the start of the run (as specified by the parameter number), and the maximum number of cells allowed in a multicellular organism (as specified by the parameter max_cells_per_organism); and the second line is a separator. Each subsequent line of the file is of the format:

```
TimeSliceNumber NewSpecies OrganismBirths CellSplitInfo CellDeathInfo
OrganismFissionInfo OrganismMovementInfo
```

where `NewSpecies` is the number of new species that have appeared in the population in the period since the previous line of the file was recorded; `OrganismBirths` is the number of new organisms that were born in that period; `CellSplitInfo` shows the number of organisms which grew in size (by executing the instruction `nwm_split`), followed by a list of the size (in terms of number of cells—before the cell division) of each such organism; `CellDeathInfo` shows the number of cells that died, followed by a list of the size (before the cell death) of the organism to which each one belonged; `OrganismFissionInfo` shows the number of organisms which fissioned (due to cell death), followed, for each one, by the number of fragments (new organisms) that resulted from the fission, and a list of the size of each new organism; and the final block of information, `OrganismMovementInfo`, shows the number of organisms which moved, followed, for each one, by a triplet of numbers indicating the size of the organism, and the x and y components of its movement.

E.2.7 Backup File (`autosave.ser`)

Cosmos records its state at regular intervals during a run, so that in the event of a run being stopped prematurely, it can be restarted from the last saved position. The time period between saves is set by the parameter `backup_period`. The data is recorded to the file `autosave.ser`. Should a run need to be restarted from this file, it should be placed in the input directory, and the system started with the parameter `restart` set to `yes`. The format of the saved data is somewhat complicated, but the user should not need to worry about this. Occasionally, however, it may be desirable to extract data from this file, as it contains a complete snapshot of the run at the given time. If this is necessary, the format can be ascertained by studying the `Serialise` method of the class CM_Process, in the source file `Process.cc`.

E.2.8 Visualisation Output Files

If the parameter `visualisation_recording_on` is set to `yes`, various kinds of data are written to files for subsequent playback as 'movies' of the run. Each file contains data about the spatial distribution of a particular aspect of the system, at a number of times during the run.

There are seven different aspects of the system that can be recorded in this way. These are:

1. The ages of the cells in the population, expressed as the number of time slices that have elapsed since their birth (recorded in the file `v_age.dat`).
2. The number of energy tokens that each cell has in its Energy Token Store (recorded in the file `v_cell_energy.dat`).
3. A flag to indicate whether each cell has executed any foreign code from its Received Message Store during its lifetime (recorded in the file `v_comms.dat`).
4. The number of energy tokens stored at each square in the environment (recorded in the file `v_env_energy.dat`).
5. The SpeciesID of each cell, which also indicates the length of each cell's genome (recorded in the file `v_id.dat`).

6. The size of each organism, in terms of number of cells (recorded in the file `v_orgsize.dat`).
7. The direction of movement (if any) of each organism in the previous time slice (recorded in the file `v_move.dat`).

Normally, if `visualisation_recording_on` is set to `yes`, then all of these files get written. However, if `visualisation_record_energy_only` is additionally set to `yes` rather than `no`, then only the files `v_cell_energy.dat` and `v_env_energy.dat` are written.

The files are updated during the run at intervals determined by the system parameters `visualisation_intersample_period`, `visualisation_intrasample_period` and `visualis ation_sample_size`. Specifically, data is recorded for a number of sample periods during the run. The number of time slices between successive samples is determined by `visualisation_ intersample_period`. Each sample consists of data for a number of time slices, determined by `visualisation_sample_size`. The number of time slices between successive records within a sample is determined by the parameter `visualisation_intrasample_period`.

At each time slice when a record is to be made, a batch of data is written to each file. This data is written in `grid_size`+1 rows of `grid_size`+1 columns. The elements of the final row, and of the final column, are all `-1`. This extra row and column is added purely to easy the process of producing a graphical display from the data using the MATLAB visualisation software package. The remaining elements of the data correspond to individual squares of the environment. For the file `v_env_energy.dat`, each element represents the number of energy tokens available at the corresponding square. For the other files, the element represents data associated with any cell(s) that are present at the corresponding square. If no cells are present, the element is given the value `-1`. If a single cell is present, the element is given the appropriate value (according to which file is being written) for that cell. If multiple cells are present, the element contains the appropriate values for each cell, separated by colons (`:`s).

An extra file, `v_idx.dat`, is also written along with these other visualisation files. At each time slice when data is written to the other files, the corresponding time slice number is written to `v_idx.dat`.

F Implementation Details

The core of the Cosmos system and REPLiCa programming language is implemented as an object-oriented system in ANSI standard C++ (with heavy use of the C++ Standard Template Library). It is compiled with the GNU C++ complier in a Unix (Solaris) environment, but should be portable to other compilers and platforms.

Cosmos uses the bsd_random() pseudo-random number generator (RNG), which uses the linear feedback shift register generation technique. bsd_random() does not suffer from some of the deficiencies of many versions of the standard random() RNG.

References

[Adami & Brown 94] Chris Adami and C. Titus Brown. Evolutionary learning in the 2D artificial life system 'Avida'. In R Brooks and P Maes, editors, *Artificial Life IV*, pages 377–381. The MIT Press, 1994.

[Matthews 91] R.E.F. Matthews. *Plant Virology.* Academic Press, San Diego, CA, 3rd edition, 1991.

[Ray 91] Thomas S. Ray. An approach to the synthesis of life. In C.G. Langton, C. Taylor, J.D. Farmer, and S. Rasmussen, editors, *Artificial Life II*, pages 371–408. Addison-Wesley, Redwood City, CA, 1991.

[Taylor & Hallam 97] Tim Taylor and John Hallam. Studying evolution with self-replicating computer programs. In P. Husbands and I. Harvey, editors, *Fourth European Conference on Artificial Life*, pages 550–559. MIT Press/Bradford Books, 1997.

[Taylor 96] TJ Taylor. The COSMOS environment and REPLiCa programming language. Departmental Working Paper No. 259, Department of Artificial Intelligence, University of Edinburgh, June 1996.

[Taylor 99] Tim Taylor. *From Artificial Evolution to Artificial Life.* Unpublished PhD thesis, Division of Informatics, University of Edinburgh, 1999.

[Thearling & Ray 94] Kurt Thearling and Thomas S. Ray. Evolving multi-cellular artificial life. In R. Brooks and P. Maes, editors, *Artificial Life IV*, pages 283–288. The MIT Press, 1994.

[Thearling 94] Kurt Thearling. Evolution, entropy and parallel computation. In W. Porod, editor, *Proceedings of the Workshop on Physics and Computation (PhysComp94)*, Los Alamitos, November 1994. IEEE Press.