The COSMOS Environment and REPLiCa Programming Language

DAI WORKING PAPER No. 259

Tim Taylor
(E-mail: timt@aifh.ed.ac.uk)

Department of Artificial Intelligence
University of Edinburgh
June 1996

The COSMOS Environment and REPLiCa Programming Language

Tim Taylor
Department of Artificial Intelligence,
University of Edinburgh,
5 Forrest Hill, Edinburgh EH1 2QL, Scotland.
E-mail: timt@dai.ed.ac.uk

4 June 1996

1 Introduction

This paper describes the system that I am developing to study the open-ended evolution of parallel computer programs. The system is called `COSMOS` (standing for COmpetitive Self-replicating Multicellular Organisms in Software). The general design of `COSMOS` is described first, followed by details of the programming language (called `REPLiCa`) in which the programs are written. A detailed explanation of the `COSMOS` system is then given, followed by a list of suggested experiments. Finally, a discussion of the differences between `COSMOS` and Tom Ray's `Tierra` system is presented.

2 General Overview

`COSMOS` is an environment that facilitates the study of patterns of evolution among self-replicating entities which are competing with each other for resources. In this environment, computer programs are the self-replicators, and large numbers of them compete with each other for the memory and CPU time required to make copies of themselves. The programs are also subject to mutation, so that, over time, mutants which are better at making copies of themselves become more numerous in the population of programs, and a process of evolution is observed.

I consider the serial self-replicating programs with which the system is primed to be somewhat analogous to the unicellular eukaryotic biological organisms that are thought to have been abundant on Earth prior in the Vendian period of geological time, immediately before the emergence in the fossil record of macroscopic multicellular organisms and the 'Cambrian explosion' that followed. Following this analogy, multicellular organisms may be compared to parallel programs in my system. Instructions are included in the `REPLiCa` language to enable a program to dynamically create multiple processes, so that evolution can explore the possibilities of parallelism (multicellularity).

The main purpose of `COSMOS` is to provide another tool for studying general questions about evolution. The more such tools we have to supplement what may be learned from biological evolution on Earth, the more we will be able to refine our understanding of evolutionary genetics and the dynamics of evolutionary systems in general. Additionally, an analysis of the evolved programs may suggest novel forms of parallelisation of computer algorithms. My motivations for developing `COSMOS` are discussed in more detail in [Taylor 96a] and [Taylor 96b].

Before continuing, I should clarify some of the terms that will be used throughout the paper. I shall tend to use biological terms when describing COSMOS because these tend to be somewhat shorter than the associated computer terms; this is just to make the paper easier to read. While these biological terms suggest the analogy that I had in mind when designing COSMOS, 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 following table lists the meanings I wish to attach to some biological terms in the present context.

Term	*Meaning in context of* COSMOS
Genotype	The instructions that make up a program (the host code within a cell).
Phenotype	The action (behaviour) of a progam as its instructions are being executed.
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 and registers.
Unicellular	An organism containing a single cell (process), i.e. a serial program.
Multicellular	An organism containing multiple cells (processes), i.e. a parallel program.
Organism	A single program, which may be unicellular or multicellular.

Table 1: Definitions of some terms used in this paper.

COSMOS bears some similarity to Tom Ray's Tierra system [Ray 91] [Ray 94], but, as mentioned already, it has been designed specifically to look at the evolution of parallel programs. A more detailed comparison between the two systems is given in Section 8.

Some work has been done with evolving parallel programs in Tierra [Thearling & Ray 94] [Thearling 94][1], but this has used a shared memory model of parallelism, and experiments so far have resulted in the evolution of only SIMD (Single Instruction, Multiple Data) programs. In contrast, COSMOS uses a distributed memory model, and has design features inspired by biological analogy which should make the evolution of MIMD (Multiple Instruction, Multiple Data) programs much more likely. This being the case, COSMOS has to rely heavily on communication between processes to perform parallel computations. In fact, message passing is the only way that one cell within a multicellular organism can communicate with its neighbours, or with other organisms. Apart from this intercellular communication, each cell only has read, write and execute access within its own cell boundary, which again is fundamentally different to Tierra and similar systems (e.g. [Skipper 92] [Koza 94] [Adami & Brown 94] [Pargellis 96]).

There are also a number of other fairly important features designed to encourage the evolution of *diversity* and *complexity*[2] in the competing programs, rather than just the optimisation of

[1]Throughout this paper I will refer to this work as Parallel Tierra

[2]The word "complexity" is used in an informal sense here, not least because there is little agreement on a satisfactory formal definition of the term when applied to evolving systems. When using this word I am referring to complexity at the level of the multicellular organism, and believe that the emergence of such complexity relies on selective forces acting on organisms which possess a developmental process from singled celled zygote

their ancestral algorithms. The most important of these are the energy token allocation system, described in the Section 4, and the *tag* system. The tag system is closely linked to the programming language, `REPLiCa`, in which the self-replicators are written. This language is introduced in Section 3.

3 The `REPLiCa` Programming Language

The self-replicating programs in `COSMOS` are written in a programming language called `REPLiCa` (standing for Robust Evolvable Production Language for Cosmos). An annotated list of the `REPLiCa` instruction set can be found in Appendix A.

A `REPLiCa` program is not in the format of traditional computer code where execution starts at the beginning and proceeds until the end of the code (ignoring features such as jumps and conditional branching). Rather, it consists of a sequence of *tag blocks*. The structure of a typical section of `REPLiCa` code is shown in Figure 1. The mechanics governing the flow of control in a `REPLiCa` program are somewhat similar to those of a Production System [Minsky 67] or the Rule and Message component of a Classifier System [Goldberg 89]. (It is for this reason that I have used the term *Production Language* in the name of the language.)

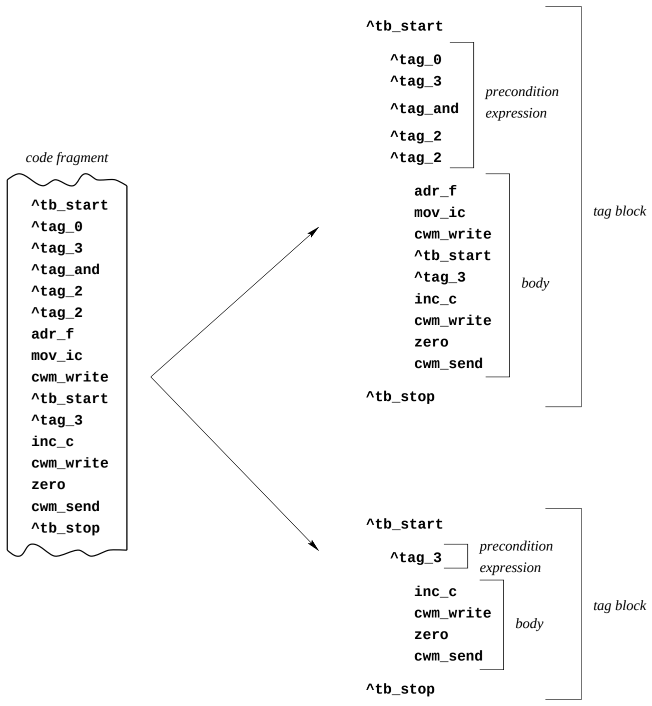

Figure 1: The structure of a typical section of `REPLiCa` code. This code fragment contains one tag block embedded within another one.

A *tag block* starts with a `^tb_start`[3] instruction and ends with a `^tb_stop`. (In this way, one tag block may actually be embedded within another if two `^tb_start` instructions occur before

to multicellular adult, and on exploitative co-evolution between different species. An important goal of my early research programme will be to decide upon a suitable way of quantifying such complexity.

[3]Instructions which begin with the ^ symbol are actually meta-instructions. They are not executed at runtime like other instructions, but instead provide information about the structure of the code and of tag blocks and templates (as explained in the main text).

a `^tb_stop`—and similarly for three or more `^tb_starts`. In this situation, the first occurrence of a `^tb_stop` instruction marks the end of *both* tag blocks.) Immediately after the `^tb_start` instruction there may be a *precondition expression* which must be satisfied before that tag block can be executed. If no precondition expression follows the `^tb_start`, the tag block can never be activated.

A *precondition expression* is a sequence of *tag patterns* separated by instructions taken from the set {`^tag_and`, `^tag_or`, `^tag_not`}.

A *tag pattern* is a continuous sequence of 1, 2, 3 or 4 instructions, taken from the set {`^tag_0`, `^tag_1`, `^tag_2`, `^tag_3`, `^tag_#`}. If a tag pattern appears which is more than four instructions long, the extra instructions are ignored.

Note that, with a system such as this, sections of code which are not surrounded by a `^tag_start` and `^tag_stop` will never be executed (i.e. a program may contain introns, to borrow another biological term).

Any `REPLiCa` program has an associated *tag store* containing a number of *tags* at various concentrations. The tag store performs a similar function to the message list of a classifier system

A single *tag* is similar to a *tag pattern* but may not contain the symbol `^tag_#`. In other words, a tag is a continuous sequence of between 1 and 4 instructions taken from the set {`^tag_0`, `^tag_1`, `^tag_2`, `^tag_3`}. This therefore allows for $4^4 + 4^3 + 4^2 + 4^1 = 340$ unique tags.

Tags may be produced as the code is executed (using the `ts_create` command), and are stored in the tag store. Multiple copies of any particular tag may be present in the store—that is, each tag in the store has a concentration associated with it. Equally, tags may be removed from the tag store (using the `ts_destroy` command), and their concentrations in the store are also subject to natural decline.

The tag patterns in precondition expressions on the code refer to the tags in this store, with the `^tag_#` symbol acting as a single wildcard character. Execution of a particular tag block depends upon the presence and/or absence of tags in the store as dictated by the block's tag expression, and, if more than one tag block has a satisfied tag expression, on the relative concentrations of the relevant tags in the store. After the selected tag block has been executed, the tag expressions of all tag blocks are then re-evaluated in order to find the next block to be run.

The precise mechanisms for governing flow of control in this system will be ironed out in the initial stage of this project. During this stage, a basic `COSMOS` interpreter will be written to enable the study of the dynamics of this type of language. In order not to constrain the evolutionary potential of the system, it is of prime importance to ensure that `REPLiCa` is computationally complete (as has been proven for the `Tierran` language [Maley 94]).

During this initial stage, the following aspects of the language will be investigated:

- Allowing tags in the tag store to simply take on a binary value of *present* or *absent* rather than having an associated concentration as described in this section.

- The effect of varying the rate at which the concentrations of tags in the tag store naturally diminish (i.e. decline automatically over time as opposed to be actively removed by the `ts_destroy` command). This will include running the system with *no* natural decline of tag concentrations.

- Various conflict resolution mechanisms for cases where more than one tag block have satisfied precondition expressions.

- The effect of giving individual tag blocks a priority (specified with the precondition expression) which is taken into account by the conflict resolution mechanism.
- The effect of *preemptive* switching of tag blocks (i.e. re-evaluating tag block precondition expressions as soon as there is any change in the contents of the tag store, and, if appropriate, switching to a new tag block immediately rather than allowing the current block to complete execution).

The tag instructions are also used as templates for marking and locating sections of code. The instructions `adr`, `adrf` and `adrb` take a template (which is of the same form as a tag pattern), and look for a complementary pattern of instructions in the code. The complement of a `^tag_0` instruction is a `^tag_1`, and the complement of a `^tag_2` instruction is a `^tag_3`. If the wildcard instruction (`^tag_#`) is used in the template, it will match any of these four tag instructions. For example, the code

```
adrf, ^tag_0, ^tag_3, ^tag_#, ^tag_2
```

would search forward from the current read position until it found the following sequence of instructions:

```
^tag_1, ^tag_2, {^tag_0,^tag_1,^tag_2,^tag_3}, ^tag_3
```

If successful, the address of the last of these instructions is placed in the `AX` register, otherwise the flag is set.

4 Actions and Interactions

The `COSMOS` memory is distributed across a number of processors, each of which may be real or virtual (i.e. simulated on a serial processor). As evolution proceeds, the memory becomes filled with cells, some of which may be independent organisms (i.e. serial programs), and others may be components of multicellular organisms (i.e. parallel programs). The structure and dynamics of a single cell are now described, followed by a description of intercellular and inter-organism communications.

4.1 The Structure and Dynamics of a Cell

The basic structure of a single cell is shown diagrammatically in Figure 2. Each cell is a virtual CPU (vCPU), having its own program code, working memory, stack, registers and various other structures.

Some notable features of the cell are as follows:

4.1.1 Tag System

The basic mechanics of the tag system were described in Section 3. The only point to be added now is that a cell in a multicellular organism can send tags from its tag store to neighbouring cells in the organism (using the `ts_send` command). In this way, the behaviour of any cell in the organism is affected by the behaviour of its neighbours. For more details about intercellular communications, see Section 4.2.1.

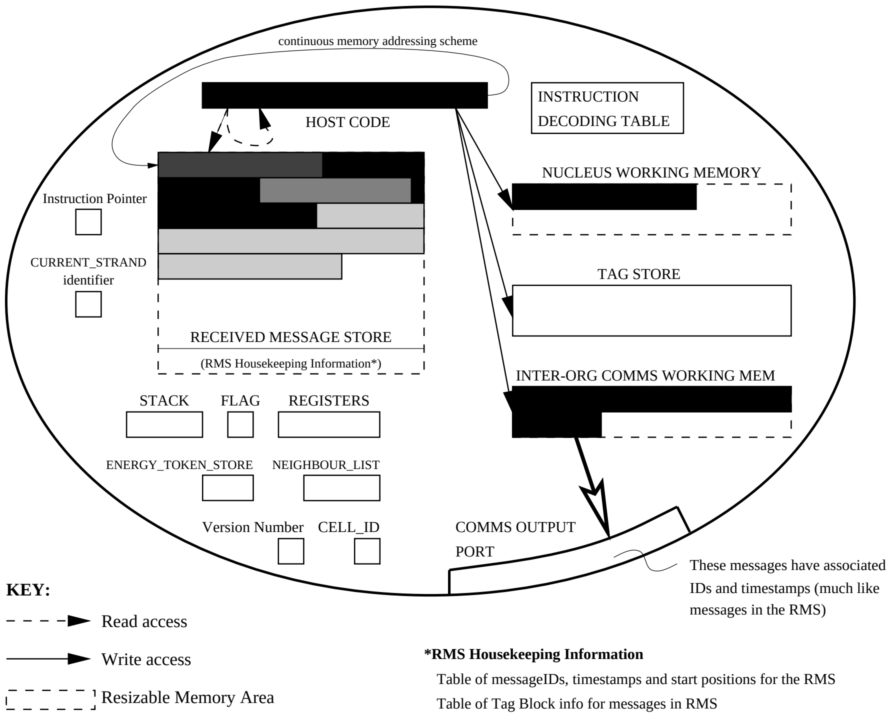
continuous memory addressing scheme
HOST CODE
INSTRUCTION DECODING TABLE
NUCLEUS WORKING MEMORY
Instruction Pointer
CURRENT_STRAND identifier
TAG STORE
RECEIVED MESSAGE STORE
(RMS Housekeeping Information*)
INTER-ORG COMMS WORKING MEM
STACK
FLAG
REGISTERS
ENERGY_TOKEN_STORE
NEIGHBOUR_LIST
Version Number
CELL_ID
COMMS OUTPUT PORT
These messages have associated IDs and timestamps (much like messages in the RMS)
KEY:
Read access
Write access
Resizable Memory Area
*RMS Housekeeping Information
Table of messageIDs, timestamps and start positions for the RMS
Table of Tag Block info for messages in RMS

Figure 2: The structure of a cell in COSMOS.

4.1.2 Energy Token Store

A large number of cells may exist concurrently in the memory of a given processor. In order to run the code of all of these cells, the processor must time-slice between each cell. In COSMOS, unlike in other systems, each cell is allowed to execute a fixed number of instructions during each time slice. This number is the same for all cells, and cannot be varied.

To execute its instructions when a cell receives a time slice, it must pay one energy token to the processor. A cell has a store of energy tokens (which it collects from the environment as described shortly). If the number of tokens in this store falls below a particular threshold, the cell dies. Additionally, when the amount of free memory available in a given processor is low, the processor will kill off a number of cells with the smallest number of energy tokens, 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.

After the processor has finished each time-slice sweep across all of its cells, it releases a certain number of energy tokens into the environment[4]. These tokens are then available to be collected by cells, by the use of the `et_collect` command.

A multicellular organism may also pass energy tokens between its cells (using the `et_transfer` command), 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, it is hoped that programs may 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 com-

[4]Actually, this is not quite the full story. A time-slice sweep is coordinated across a group of *virtual processors* called a *locality* (see Section 4.5). Also, energy tokens are distributed in a place (the *Daemon Virtual Processor*, Section 4.5.2) where all cells in the locality may collect them.

plicated 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 environment (where they may be collected by other organisms). This mechanism opens the possibility of one type of organism evolving the ability to 'kill' members of another type and to collect the energy tokens thus released into the environment. The prey organisms would presumably evolve measures to prevent this, leading to a process of exploitative co-evolution [Maynard Smith 89]. Of course, this is just speculation about a possible scenario. Whether such interactions will actually be observed in the system remains to be seen, but even if they did, they would presumably take a *very* long time to emerge.

4.1.3 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.

Rather than directly writing instructions one at a time to a new area of memory, 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` command. These have the effect of transferring the contents of the nucleus working memory into a new cell in the locality[5]. The former command 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).

In either case, upon division the contents of the energy token store and the tag store are divided equally between parent and child cell. The received message store, inter-organism communications working memory and the nucleus working memory of the new child cell are initially empty.

Note that the parent cell has no control over the exact memory location in which the child cell is placed, or even of the virtual processor upon which it will be placed (see Section 4.5). Such memory management is the responsibility of the `COSMOS` operating system. This is a general feature of the system; cells do not concern themselves with the details of memory management, and indeed have no concept of their location, or that of other cells, within the memory of a processor or locality.

4.1.4 Inter-Organism Communication Structures

Two major cell structures remain to be explained; these are the received message store (RMS) 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 4.2, but the action of messages which have arrived in the RMS is explained now.

Inter-organism messages take the form of strings of instructions. Each individual message is also marked by a messageID defined by the originator of the message. A cell may choose

[5]See Section 4.5 for information about localities.

to receive all messages in the locality which have a given messageID (using the `rms_receive` command). If any such messages are available, they are copied into the cell's RMS.

The host cell may process these messages, using the `str_switch` and `adr` commands to set the `AX` register to an address within a message, and using commands such as `movic` and `inc_a` to sequentially read the message.

Messages in the RMS 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 messageID number. The host code of the cell also has a cellID number associated with it[6]. If any messages in the RMS happen to have a messageID which matches the cellID of the host code, then it is treated equally to the host code in that, if the message contains tag blocks with satisfied precondition expressions, then the code in the tag block may be executed within the host cell. A cell has several lines of defence against such parasitism, which will be mentioned in Section 4.2.2.

4.2 Intercellular and Inter-Organism Communications

There are two distinct types of communication that cells may employ. One is used for communicating with other cells within the same organism, and the other is used for communicating with other organisms.

The general philosophy governing the design of these communication facilities has been to provide the organisms with as rich an environment as possible. In particular, the inter-organism communications commands allow organisms to exchange arbitrary messages. It is almost certainly the case that we will not observe the evolution of organisms which take full advantage of these facilities, or which even make limited use of them, for a very long time indeed. 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.

4.2.1 Intercellular Communications

As mentioned in Section 4.1.1, a cell which is a member of a multicellular organism can communicate with other cells in the organism by sending tags from its store (using the `ts_send` command). In this way, the execution of code in a particular cell may be influenced by many other cells in the organism, because tags which are sent from one cell to another will influence which tag blocks get executed in both of the cells. Therefore, although each cell in a multicellular organism has the same genome, each cell may be executing different parts of this genome at any given time. Thus the cells of a multicellular organism are able to operate in a MIMD fashion.

A cell may not be able to communicate directly with *every* other cell in the organism, especially if the organism is large. Rather, each cell keeps a (finite) list of its 'neighbours', i.e. the cells with which it can exchange tags. As an organism grows, each cell's *Neighbour List* will link it to a number of its ancestors and a number of its descendants (in an ontogenetic sense). However, a cell's 'position' within the organism (i.e. its list of neighbours) is not set in stone during the process of development—the `migrate` instruction allows a cell to move within the organism by swapping its Neighbour List with that of another cell.

[6]The CellID is a parameter of the cell which 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. Therefore, it is possible for organisms with different cellIDs to emerge in the system.

If a cell dies, its neighbours are informed. The links to the dead cell are then removed from each of their Neighbour Lists, and replaced by new links to nearby cells.

4.2.2 Inter-Organism Communications

Inter-organism communications were introduced in Section 4.1.4. A cell can broadcast an arbitrary message to other cells in the same locality. A message is just a collection of instructions of arbitrary length, which has been composed in the inter-organism communications working memory using the command `cwm_write`. When a cell broadcasts such a message (using the `cwm_send` command) a messageID number (equal to the current value of the `CX` register) is attached to it, along with a time-stamp. As soon as `cwm_send` is called, the message is transferred to the cell's communications output port, where it remains until it is overwritten by future messages.

The complementary process of receiving messages from other cells is achieved by the command `rms_receive`. The value of the `CX` register at the time this command is issued indicates the type of message the cell will receive. When the cell issues this command, any messages which are present in the communications output ports of any cells in the same locality which have messageIDs matching this `CX` value will be copied into the receiving cell's RMS. If there is not enough free space in this store, some of the existing contents will be overwritten, starting with the oldest messages. If the new messages themselves take up more room than is available in the store, only the newest of the incoming messages will be received. When new messages are copied into the RMS, their messageIDs are also stored in the host cell, together with a time-stamp indicating when the message was received.

Once the messages have arrived in the received message store, they may be read by the host code, and some may even be treated as executable code, as described in Section 4.1.4.

Allowing organisms to exchange arbitrary genetic code has little 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.

Having said this, if a `COSMOS` cell receives foreign code which happens to have the same messageID as its host code's cellID, then the foreign code will be treated as if it is part of the host genome, as explained previously. 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 will be expected to evolve measures to prevent the foreign code from being executed. This may be achieved in a number of different ways, such as by using a different cellID for its own genome, by removing the foreign code from its store (either by directly removing it using the `str_remove` command, or by indirectly causing its removal by shrinking the size of the received message store), 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.

4.3 Environmental Information

As well as carrying specific inter-organism communications as detailed in Section 4.1.4, the environment also carries summary information about each organism in the locality. These messages are transmitted by the locality's daemon virtual processor (DVP)[7] and may be intercepted by cells in exactly the same way as they intercept other inter-organism communications. These messages contain the following information:

- The cellID of the organism's original root cell
- The number of cells in the organism
- The total number of energy tokens stored in the organism

This information is encoded in a standard way for all organisms and across all localities. The DVP updates the transmitted information at regular intervals. All of these messages have a special messageID number (which is standard across all organisms), and they may be picked up by any other cell using the `rms_receive` command.

4.4 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 genotypic novelty for selection to work upon. Mutations occur naturally throughout the system at a low rate, and may affect any section of code in any cell in the system, causing a random change in that section. In addition, variety may also be introduced into an organism by the flawed execution of an instruction in its genome. In other words, 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, the net results of mutations and flaws 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` command, 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), 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.

`Tierra` features both mutations and flaws, 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]. The effects of different rates of both mutations and flaws will be investigated for the `COSMOS` system.

4.5 Logical topology of cells and organisms

The `COSMOS` system can run across a number of virtual processors (VPs)—the term *virtual processor* is used because more than one of them may be simulated on a single physical processor if desired. Each VP has an area of RAM in which cells can live and reproduce. This section describes how the VPs within the system are grouped together, and the types of interaction permissible within and between groups of VPs.

4.5.1 Localities

VPs are grouped together into *localities* (see Figure 3). `COSMOS` can be run with any number of localities. A single locality is a collection of VPs on which organisms can grow, communicate and replicate freely. As far as an individual organism is concerned, the combined RAM of all

[7]Daemon Virtual Processors (DVPs) are described in Section 4.5.2.

of the VPs in the locality is a single, continuous and homogeneous resource within which it competes with other organisms.

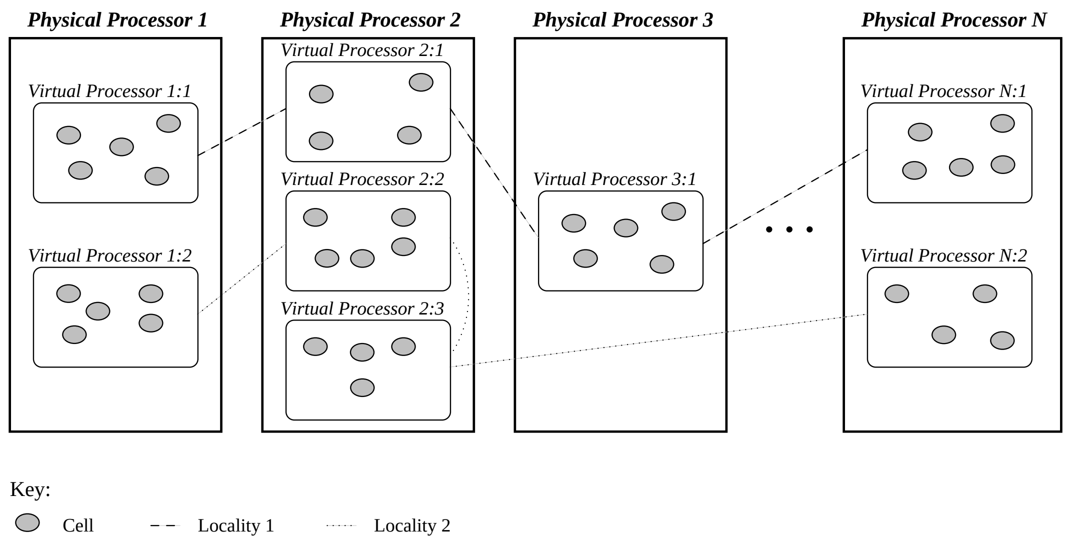

Figure 3: Example COSMOS Configuration, Indicating the Relationship between Cells, Virtual Processors and Physical Processors.

There is very little interaction between different localities. In this way, COSMOS allows evolution to proceed simultaneously across a number of 'geographically isolated' environments, thereby maintaining heterogeneity in the system as a whole. It has been suggested that such localisation may play an important role in preventing premature equilibration in systems such as this, and in preventing them from becoming trapped in meta-stable states [Adami & Brown 94].

The only time when information may be exchanged between localities is when a cell issues a `nwm_divide` command. Normally, this will result in a new child organism appearing in the same locality as the parent. However, this command will occasionally place the child cell into a different locality at random. In this way, an organism's descendants may end up in many different localities, although the rate of dissemination is slow.

4.5.2 The Daemon Virtual Processor (DVP)

Although COSMOS has been designed so that cells and organisms are as self-contained and autonomous as possible, it is necessary to keep some central store of information in each locality for the efficient implementation of various higher-level features of the system. Therefore, each locality has one VP which is dedicated to performing various housekeeping duties for the organisms within the locality. This VP is known as the Daemon Virtual Processor (DVP).

The DVP maintains a database of each cell and each organism in the locality. This information is used for a number of purposes, such as ensuring the consistency of cell numbering across all VPs in the locality, exporting summary information about the state of the locality to files for later analysis, etc.

Additionally, the DVP keeps track of the messageIDs and time-stamps of all messages currently waiting in the Communications Output Ports of each cell in the locality, thereby improving the efficiency of inter-organism communications.

The DVP also has its own Communications Output Port, much like that of an ordinary cell.

It is the responsibility of the DVP to periodically place summary information into this port about each organism in the locality (see Section 4.3).

Finally, it is the responsibility of the DVP to distribute energy tokens into the environment at the end of each time-slice sweep, and to coordinate time-slicing across all of the VPs in the locality.

5 Analysis

The core COSMOS system will be a stand-alone application. In order to allow the analysis of an evolutionary run, a file containing information about the different organisms will be written by the system during the run. The information will contain data such as the genotype of an organism, when it was created, in which locality it existed, a reference to its immediate ancestor, the time taken to reproduce etc., as well as information about the physical configuration of processors and localities. In order to lower the total amount of data produced, organisms may only be logged in this way when their numbers exceed a certain threshold proportion of the total population of the system. This log file will enable detailed analysis of a run to be conducted at a later date, and will also provide a way of restarting the COSMOS system in the event of a crash.

A separate visual analysis application will also be written, which will enable a user to inspect an evolutionary run while it is in progress. This will be achieved by displaying data from the log file in a variety of ways. This viewer application may also allow a user to interactively change parameters and organisms in the middle of a run[8].

6 Global Parameters

The COSMOS system as described contains a number of global parameters, as listed below (in no particular order). The sensitivity of the system to changes in the values of these parameters will be studied.

- The probability of a `nwm_send` instruction placing the new organism in a different locality to the parent.
- The length of a cell's Neighbour List (i.e. the number of neighbouring cells in a multicellular organism that a cell can directly influence by the transmission of tags).
- The maximum length of a tag, which of course dictates how many unique tags may exist in the tag store.
- The mutation rate.
- The flaw rate for execution of instructions.
- The number of energy tokens placed in the environment by the DVP after each time-slice sweep.
- The number of instructions that a cell can execute in exchange for a single energy token.
- The number of instructions that a cell is allowed to execute during one time-slice.

[8]Note, however, that it has not been decided whether this feature will definitely be included at present, as the wisdom of interactively interfering with a particular evolutionary run is less than clear.

- The number of bytes by which a buffer is increased or decreased by the execution of a single <buffer>`_inc` of <buffer>`_dec` command.
- The number of energy tokens expended or reclaimed by the execution of a single <buffer>`_inc` of <buffer>`_dec` command (but see points in next section as well).
- The threshold of free memory in a locality below which existing cells start to get killed off to make more room.
- The maximum concentration of a single tag in the tag store of a cell.
- The initial sizes of the Nucleus Working Memory, the Inter-Organism Communications Working Memory and the Received Message Store.
- The threshold number of stored energy tokens below which a cell dies.

7 Some Suggested Experiments

There are a number of fairly arbitrary aspects of the design of `COSMOS`. These should be experimented with in order to test their effects on the observed results.

- Details of the tag system, as listed at the end of Section 3. Additionally, there is a question of whether the `ts_send` command sends a tag from the tag store, or creates a new tag and sends it without changing the contents of the store.
- Details of the division of a cell's resources following the execution of a `nwm_split` or a `nwm_divide` command. In particular, the effect of allowing energy tokens and tag store contents to be divided unequally between mother and child cells will be investigated. If, for example, the energy tokens were divided unequally, a mother cell may build up a large collection of them which are then nearly all donated to the child organism following a `nwm_divide` command, thus giving the child extra energy that it can use for development before having to collect energy tokens for itself. In other words, this scheme would allow for the evolution of organisms which hatched their offspring in an 'egg' with a plentiful food supply.
- Allowing a cell to change the size of its Neighbour List, so that it can influence (and be influenced by) more or less cells in the organism.
- The order in which cells in a locality are given their time-slice during one time-slice sweep across all the cells. Several random and non-random alternatives will be tested.
- The number of different tag symbols. The system as described has four tag symbols (`^tag_0`, `^tag_1`, `^tag_2`, `^tag_3`). The effect of varying this number will be studied.
- Details of cell death due to overcrowding. When the amount of free memory in a locality falls below some threshold value, cells are killed off depending on how many energy tokens they have stored. This selection may be deterministic or stochastic. (Stochastic selection will probably be required to avoid the possibility of immortal organisms—see the point on "Organism Death" in the following list.)
- Details of the mechanism whereby a cell may increase or decrease the size of its memory areas. The effect of imposing an on-going (rather than one-off) cost (in the form of increased energy token expenditure) will be studied.

- The `REPLiCa` instruction set. A number of instructions (denoted with square brackets [] in Appendix A) may or may not be included in the set.

In addition to investigating parameter sensitivity and specific aspects of the design, a number of more general questions may be studied with `COSMOS` which may also have a bearing on biological evolution. A few of these are listed below.

- **Speciation**. Do different types of program emerge which are capable of coexistence?

- **Multicellularity and Organism Life Cycles**. Do multicellular organisms emerge, and are they evolutionarily stable? What types of life cycle do they adopt? (e.g. Do they have an initial period of growth followed by a relatively stable adult stage?, How large are these organisms? How much diversity is there in size between different types of organism? What types of processes are occurring within these multicellular organisms? Do individual cells within an organism become specialised to particular tasks?).

- **Organism Death**. Depending on how the parameters are set up, cells (and organisms) may potentially last forever[9] if the rate at which they collect energy tokens from the environment is high enough. It will be of interest to observe whether immortal programs do emerge, or whether evolution can dictate that there is an optimal life span which is a compromise between replicator longevity and the evolvability of the lineage. In other words, is organism death an inevitable feature of evolution in this type of system? It will also be interesting to see whether `COSMOS` is capable of maintaining organisms which have a wide variety of lifespans.

- **Change in the Complexity of Organisms** The question of whether the process of evolution is sufficient for the production of complex organisms is currently receiving much interest from researchers across a number of disciplines (e.g. [Dawkins 86], [Kauffman 93], [Heylighen 96]). As mentioned earlier in this paper, it is first necessary to decide upon a formal definition (or various definitions) of complexity (at least as applied to evolution) before this question may sensibly be considered. My work will include a survey of current ideas about the quantification of complexity in evolution, and the application of some of these methods to the analysis of evolutionary runs on `COSMOS`. I am particularly interested in the emergence of interactions between organisms and other actions of an organism which are not *directly* related to its self-replication.

- **Co-evolution among different Species**. As I have briefly mentioned previously, the evolution of complexity is thought to be closely related to co-evolution among different organisms. Co-evolution may be divided into three different scenarios; competitive, exploitative and mutualistic [Maynard Smith 89]. Analysis of `COSMOS` runs will reveal which, if any, of these processes are occurring, and will hopefully provide clues as to *why* they are or are not occurring.

- **Evolution of the Genetic Code**. Experiments will be conducted in which the actual genetic code which determines the decoding of a pattern of bits in the cell's genome into a string of instructions is itself evolvable. This would entail equipping each cell with a map describing the code.

[9]Actually, if cell death caused by overcrowding is a stochastic process, there is always a chance that *any* cell may be killed. Therefore, no organism has guaranteed immortality.

If more bits are used to encode each instruction than are required to cover the whole instruction set, then some redundancy can be introduced into the coding. In other words, several different bit patterns may encode the same instruction. Such redundancy is, of course, also seen in the genetic code of biological organisms.

If this code were evolvable, it is possible that it may develop towards an optimal encoding where instructions which are used the most by organisms in the system are encoded with the highest redundancy, thereby increasing the chances that such instructions will persist as as genomes are subjected to mutation.

- **Use of Environmental Information**. Experimentation will include adding environmental information about the performance of each locality (e.g. average time for one time-slice sweep, number of VPs etc.) and allowing organisms to choose the locality into which they place their offspring. It is possible that in this way the organisms may be able to find an optimal distribution across all localities for a given hardware configuration.

8 Major differences between COSMOS and Parallel Tierra

1. Tag system

   The tag system of COSMOS has no equivalent in Tierra. It was 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 specialisation and MIMD parallelism. The design of the tag system was inspired by the processes of chemical signalling between cells in biological organisms.

2. CPU time allocation

   In COSMOS, each cell on a processor is allowed to execute an equal number of instructions during each time slice. However, each time slice costs the cell one *energy token* from its store. The mechanisms for energy token distribution, storage and use were described in Section 4.1.2.

   In Tierra, programs are given CPU time according to their length. A separate 'reaper queue' mechanism in employed to govern cell death. Such a mechanism is redundant in COSMOS, where cell death is dictated by the amount of energy stored in the cell. This process imposes fewer constraints on the lifespan of cells and organisms.

3. 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 RAM). A similar situation exists in COSMOS. However, Tierran programs have read and execute privileges to *all* areas of RAM, 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 4.2).

4. Exchange of messages and genetic information

   The COSMOS mechanisms for the direct exchange of messages and 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.

5. 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. The copy is then issued *en masse* to a new memory location.

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

   There isn't really a great deal of difference between the two mechanisms, but the 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 vCPUs (cells), so that a multicellular program must actually copy itself from one vCPU 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.

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

6. 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 with the use of localities, as described in Section 4.5.1.

7. Size of Instruction Set

   The REPLiCa instruction set is about twice as big as that of the Tierran language. It remains to be seen what the effects of this will be.

8. COSMOS ancestor is serial

   In Parallel Tierra, the ancestral program was itself parallel. In COSMOS it is hoped that parallel programs may evolve from an initial serial ancestor.

9. Memory Model

   COSMOS uses a distributed memory model of parallelism, and the tag system that it employs should promote the emergence of MIMD programs. Parallel Tierra uses a shared memory approach, and, although it is theoretically capable of supporting MIMD programs, has so far only demonstrated the evolution of SIMD programs.

10. Memory Addressing Scheme

    `COSMOS` cells use a local addressing scheme which applies only to code and messages belonging to them. They have no knowledge of their location (or that of other cells) in the global addressing scheme of the VP. This is in contrast to `Tierra`, which uses a global addressing scheme.

9 Implementation

The core of the `COSMOS` system and `REPLiCa` programming language will be implemented in C++, using the MPI message passing interface. The source code will (in theory) be portable. Specifically, it should compile on Sun workstations, and also on the Cray-T3D parallel computer at the Edinburgh Parallel Computing Centre (EPCC).

An independent GUI application for monitoring (and maybe interactively playing about with) `COSMOS` runs will be written in Java.

Acknowledgements

I would like to thank my supervisor, Dr John Hallam, for his valuable input and interesting discussions about the design of `COSMOS`.

References

[Adami & Brown 94] C Adami and CT 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.

[Dawkins 86] R Dawkins. *The Blind Watchmaker.* Longman, Harlow, 1986.

[Goldberg 89] DE Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Reading, MA, 1989.

[Heylighen 96] F Heylighen. The growth of structural and functional complexity during evolution. In F Heylighen and D Aerts, editors, *The Evolution of Complexity.* Kluwer Academic Publishers, 1996.

[Kauffman 93] SA Kauffman. *The Origins of Order: Self-Organization and Selection in Evolution.* Oxford University Press, 1993.

[Koza 94] JR Koza. Artificial life: Spontaneous emergence of self-replicating and evolutionary self-improving computer programs. In C Langton, editor, *Artificial Life III*, volume XVII of *Santa Fe Institute Studies in the Sciences of Complexity*, pages 225–262. Addison-Wesley, 1994.

[Maley 94] CC Maley. The computational completeness of Ray’s Tierran assembly language. In C Langton, editor, *Artificial Life III*, volume XVII of *Santa Fe Institute Studies in the Sciences of Complexity*, pages 503–514. Addison-Wesley, 1994.

[Maynard Smith 89] J Maynard Smith. *Evolutionary Genetics.* Oxford University Press, 1989.

[Minsky 67] ML Minsky. *Computation: Finite and Infinite Machines.* Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ, 1967.

[Pargellis 96] AN Pargellis. The spontaneous generation of digital 'life'. *Physica D*, 91:86–96, 1996.

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

[Ray 94] TS Ray. Evolution, complexity, entropy and artificial reality. *Physica D*, 75:239–263, 1994.

[Skipper 92] J Skipper. The computer zoo—evolution in a box. In FJ Varela and P Bourgine, editors, *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*, pages 355–364, Cambridge, MA, 1992. MIT Press.

[Taylor 96a] TJ Taylor. On the incorporation of a developmental process in a system of self-replicating programs. Departmental Working Paper No. 257, Department of Artificial Intelligence, University of Edinburgh, January 1996.

[Taylor 96b] TJ Taylor. Principles for improving the evolvability of digital self-replicators. Unpublished internal paper, Department of Artificial Intelligence, University of Edinburgh, February 1996.

[Thearling & Ray 94] K Thearling and TS 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] K 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.

APPENDIX

A The REPLiCa Instruction Set

There are 4 (16 bit) registers and one flag. The registers **ax** and **bx** are used primarily for storing and manipulating addresses, whereas the registers **cx** and **dx** are used for arithmetic.

(*Square brackets [] indicate instructions which may be experimented with but which won't necessarily remain in the language in the long term.)*

- *Memory movement*

```
push_ax         ; push ax onto stack
push_bx         ; push bx onto stack
push_cx         ; push cx onto stack
push_dx         ; push dx onto stack
pop_ax          ; pop stack into ax
pop_bx          ; pop stack into bx
pop_cx          ; pop stack into cx
pop_dx          ; pop stack into dx
mov_cd          ; dx=cx
mov_ab          ; bx=ax
mov_ic          ; copy memory at address [ax] to cx
```

- *Calculation*

```
clr_f           ; flag=0
[set_f]         ; flag=1

inc_a           ; increment ax (if overflow, flag=1)
inc_c           ; increment cx (if overflow, flag=1)
dec_a           ; decrement ax (if underflow, flag=1)
dec_c           ; decrement cx (if underflow, flag=1)
sub_ab          ; cx=ax-bx
[add_ac]        ; ax=ax+cx
sub_ac          ; ax=ax-cx
add_cd          ; cx=cx+dx
sub_cd          ; cx=cx-dx
zero            ; cx=0
[not]           ; cx=(bitwise not)cx
not_lo          ; flip low bit of cx
[not_hi]        ; flip high bit of cx
shl             ; shift bits in cx left  (lo bit <- flag, hi bit -> flag)
[shr]           ; shift bits in cx right (hi bit <- flag, lo bit -> flag)
[xor]
```

- *Instruction pointer manipulation*

```
if_fl           ; if (flag=1) execute next instruction
                ; otherwise skip it

if_z            ; if (CX=0) execute next instruction
                ; otherwise skip it
```

- *Nucleus Working Memory*

```
                [ nwmWP = NWM Write Position ]

nwm_clear       ; nwmWP=0

nwm_write       ; if (nwmWP < MAXnwmWP)
                ;       CX -> [nwmWP] ; nwmWP++
                ; else
                ;       flag = 1

nwm_divide      ; Copy contents of NWM from start to nwmWP into
                ; a new cell which will become the root of a new
                ; organism. The new cell will usually be placed
                ; in the same locality as the parent, but may, with
                ; a small probability, end up on a different one.

nwm_split       ; Copy contents of NWM from start to nwmWP into
                ; a new cell which will become an additional process
                ; of the multicellular organism. Update Neighbour List
                ; if appropriate.
```

- *Tag Store*

```
ts_create       ; if ((valid tag follows instruction) &&
                ;     (conc(tag) < MAXconc(tag)))
                ;       conc(tag)++
                ; else
                ;       flag=1

ts_destroy      ; if ((valid tag follows instruction) &&
                ;     (conc(tag) > 0))
                ;       conc(tag)--
                ; else
                ;       flag=1

ts_send         ; if ((valid tag follows instruction) &&
                ;     (conc(tag) > 0))
                ;       send tag to neighbour cell indicated by lower
                ;         N bits of the CX register
                ;       conc(tag)--
                ; else
                ;       flag=1
```

- *(Inter-organism) Communications Working Memory*

```
                [ cwmWP = CWM Write Position ]

cwm_clear       ; cwmWP=0

cwm_write       ; if (cwmWP < MAXcwmWP)
                ;       CX -> [cwmWP]
                ;       cwmWP++
                ; else
                ;       flag=1

cwm_send        ; Send contents of CB from start to cwmWP
                ; to cell's COMMS_OUTPUT_PORT. The message is marked
                ; with an messageID equal to the current value of
                ; CX, and also with a time-stamp. If there is not
                ; enough room in port, old messages may be
                ; overwritten.
```

- *Received Message Store*

```
rms_receive     ; Receive all messages from cells in locality whose
                ; messageIDs match the value of CX
                ; if (message(s) exist with given messageID)
                ;       copy messages to RMS, marked with the
                ;       messageID and a time-stamp.
                ; else
                ;       flag=1
```

- *Energy token collection / transfer*

```
et_collect      ; if (environmental energy token available)
                ;       pick up one token
                ; else
                ;       flag=1

et_transfer     ; if (tokens(this)>MINtokens(this)
                ;       send 1 token to neighbouring cell indicated by
                ;       lower N bits of CX
                ; else
                ;       flag=1
```

- *Tags / Templates*

```
                [
                  The ^ symbol at the start of these instructions
                  indicates that these are meta-instructions (i.e. they
                  are not executed at runtime like other instructions,
                  but instead provide information about the structure
                  of the code and of tag blocks and templates).
                ]


^tb_start       ; Marks the start of a tag block. The code between
                ; this instruction and the next occurence of tb_stop
                ; can only be executed if the precondition expression
                ; immediately following tb_start is satisfied. The
                ; precondition expression is the maximum continuous
                ; block of instructions immediately after ^tb_start
                ; which consists only of ^tag_0, ^tag_1, ^tag_2,
                ; ^tag_3, ^tag_#, ^tag_and, ^tag_or and ^tag_not.

^tb_stop        ; marks the end of a tag block

^tag_0          ; these are the individual tag symbols...
^tag_1          ;
[^tag_2]        ;
[^tag_3]        ;
^tag_#          ; (this is the wildcard symbol)

^tag_and        ; and these are the logical operators used in
^tag_or         ;   precondition expressions
^tag_not
```

- *Searching for matching templates (on host or message strands)*

```
                [
                  A template has the same form as a tag pattern.
                  A complementary template has tag_0s swapped with
                  tag_1s and tag_2s swapped with tag_3s (e.g. the
                  complementary template of tag_0,tag_2,tag_3 is
                  tag_1,tag_3,tag_2).

                  The term ‘strand’ refers either to the host code
                  of a cell, or to individual messages within the
                  cell’s received message store (RMS).
                ]


[adr]           ; if ((valid template follows instruction) &&
                ;     (a complementary template is found on the
                ;      CURRENT_STRAND))
                ;       AX = address of nearest matching template
                ; else
                ;       flag=1

adrf            ; as ADR, but only searches forward from IP of
                ; current strand

adrb            ; as ADR, but only searches backwards from IP of
                ; current strand

[str_switch]    ; if (there exists a strand with ID=CX)
                ;       CURRENT_STRAND=first strand found with ID=CX
                ;       read position of strand reset to beginning
                ; else
                ;       flag=1

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

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

[str_host]      ; CURRENT_STRAND=host strand

[str_remove]    ; if (there exists a strand with ID=CX)
                ;       remove first strand found with ID=CX
                ; else
                ;       flag=1
```

- *Changing buffer sizes*

```
cwm_inc         ; if ((EnergyTokens > M [+threshold]) &&
                ;     (SpareSystemMemory > N))
                ;       increase size communications working memory
                ;       by N bytes. EnergyTokens -= M units.
                ; else
                ;       flag=1

cwm_dec         ; if (sizeof(CommWorkingMem) > N)
                ;       Decrease size of communications working memory
                ;       by N bytes. EnergyTokens += M units. If
                ;       necessary, cwmIP is also decreased.
                ; else
                ;       flag=1

rms_inc         ; as above, but for the received message store
rms_dec         ; (if necessary, stored messages are removed if there
                ;  is no longer room, starting with the oldest)

nwm_inc         ; as above, but for the nucleus working memory
```

```
nwm_dec         ; (if necessary, nwmIP is also decreased)
```

- *Migrating within a multicellular organism*

```
migrate         ; Swap information in Neighbour_List with that of the
                ; neigbouring cell specified by the low N bits of
                ; the CX register.
```

- *Killing the current process (cell)*

```
kill            ; Kill current cell. Any CPU time owned by cell
                ; is added to the locality's environmental store
```