Specification of the stringmol chemical programming language version 0.1

Technical Report Number YCS-2010-457

Simon Hickinbotham[1], Edward Clark[1], Susan Stepney[1], Tim Clarke[2], Adam Nellis[1], Mungo Pay[2], Peter Young[3]

[1]Department of Computer Science, [2]Department of Electronics, [2]Department of Biology
York Centre for Complex Systems Analysis, University of York, UK

sjh@cs.york.ac.uk www.yccsa.org

This work was supported by EPSRC grant EP/F031033/1

Abstract

This report describes in detail version 0.1 of a specification for a nonstandard computational model that emulates the reactivity of enzymes in bacterial cells. The model is an artificial chemistry in which the unique properties of each molecular species is encoded as a sequence of symbols. The model has a very simple cell-level representation consisting of a mixing volume and mixing equation. There is no distinction between genotype and phenotype, or data and program. Randomly selected molecules are subject to a stochastic binding test, in which their identifying sequences are subjected to a complementary alignment process. If the bind is successful, the molecules are combined to form a computational entity to run the reaction between them. At this point the sequence acts as a microprogram. These microprograms can perform a range of tasks, much as a chemical reaction can. Our first chemistry within this model solves the problem of designing a chemical "replicase", capable of creating copies of itself such that it can replenish a population of molecules that are subject to a decay process. This report gives detail to the model, describes the replicase molecule and its function, and shows an "invasion when rare" experiment.

1 Introduction

The traditional computational genetic algorithm that has existed since the 1960s is a very loose model of biological evolution. Much research effort has been expended in refining the process, but the basic model has remained relatively unchanged. We propose that a richer evolutionary model can deliver more powerful evolutionary processes. We seek alternative formulations of genetic algorithms in which complex genotype-phenotype (geno-pheno) interactions are used to organise the genome, control bloat, and thus deliver efficient learning systems. A good example of the phenomena we seek to emulate is the *transposon*, a genetic element which contain genes for the translocation of themselves and other genetic instructions on the genome. The challenge is to define a model that is sufficiently rich to allow such processes to arise.

Bacteria are the principal inspiration for our model, since their gene expression mechanisms are relatively well understood, their ontogeny is relatively simple, and they use horizontal gene transfer to spread a rapid response to new environmental conditions through the population. A key challenge is to create a metabolic model which is computationally efficient, yet allows the richness of geno-pheno interaction that we require. We believe that a fully functional bacterial model would be too difficult to devise *ab initio*. Instead, we concern ourselves with developing molecular analogues that have sufficient functionality to allow rich interactions between them to emerge. The model must encode metabolites and genomic material in such a way that they can interact. In this light, an artificial chemistry is an attractive solution, since it maintains an intuitive link with real biochemical systems, albeit in a simplified form. In addition, we limit the set of legal molecular-level events to simplify the design process.

In the early stages of this project [1] , we have concentrated on modelling bacterial processes via a limited series of reaction rules [1] and have applied such rule-based models to robot control [2]. We have designed a computational metabolome and implemented it as a network of reactions between a maximum of two molecular agents. This early work demonstrated that with simple computational restrictions on the types of reaction that are permitted, it is straightforward to design a gene regulatory network. To make the reaction networks evolvable, a system must be able to

[1]This report forms part of the Plazzmid project, EPSRC grant EP/F031033/1. Later versions of this report will incorporate revisions to the molecular specification.

subtly change the nodes (substrates), edges (reactions), and rates in the network, via changes in the binding, reaction and decay of the molecular structures. Given that the biological systems we are emulating are far to complex to simulate in full detail, our abstractions need to find appropriate granularity of change in the expressed proteins with the characteristics of mutation that the system possesses. Thus we must develop an artificial chemistry (AC) as the basis for the geno-pheno interaction.

We are investigating a range of ACs. There are three broad classes: abstract (molecular properties specified directly, so no analogy with shape) [3]; spatial shape-based [4]; program-based. Here we explore the program-based approach, with reference to a set of biological principles that we believe characterise biological systems. There are four components to this model:

1. A very simple cell-level architecture, more analogous to the containing vessels in Spiegelman's [5] RNA experiments than living cells, but which allows reactive agents to mix.
2. "RNA world", in which there is no distinction between genotype and phenotype - the machine acts as the blueprint for itself.
3. A "soft" binding process, based on Smith-Waterman alignments.
4. An analogue of protein folding, using computational program flow control (like AVIDA and Ray's Tierra [6, 7]) to allow reactions to be richly encoded.

We are working towards an agent-based system, where each agent is capable of emulating as many of the different molecular species seen in biology as possible: substrates, products, enzymes, proteins, RNA and DNA. We have designed a computational emulation of the biological system just described. All molecules in our system are defined in terms of a sequence of symbols. The symbols encode a set of bindings between symbols, and low-level programming instructions. A combination of the properties of these instructions and the sequence they are assembled into forms the basis of a "microprogram" that can emulate the range of protein functions we see in nature. The mixing of these molecules in a stochastic chemistry allows the overall program of the system to emerge. Key to this approach is the use of a sequence alignment algorithm for binding between individual molecules and for branching of molecular microprograms. The inexact alignments of sequcences allows rates of binding and execution of the microprograms to evolve. This report gives full detail to the microcodes, their execution, and the use of sequence alignment to implement an evolvable artificial chemistry.

We have developed the artificial chemistry with a specific design challenge in mind: Can a system be built that consists of reactive agents analogous to proteins, whose function is determined by a sequence of symbols, and which are capable of maintaining their population by constructing copies of themselves by reference to the symbol sequence? We call these agents "replicase molecules", since their function is to replicate themselves. This system has the advantage that only one kind of molecule needs to be present, but the function of the molecule captures the level of geno-pheno interaction that we will require in our bacterial model that is to follow.

2 Bioinspiration for the Stringmol domain model

2.1 The Cell model

Our cell model need not be complex, since the focus on this paper is molecular interactions. Speigelman [5] showed that evolvable systems can be run *in vitro* given appropriate raw materials. We follow this methodology for our current work *in silico*: the cell is little more than a container in which energy and raw materials are available for the evolvable reaction. The volume of the container is important since it can be used to control the probability of one molecular agent encountering another.

2.2 RNA-world chemistry

The RNA-world hypothesis proposes that early life on earth was formed from RNA, which is capable of storing information in a similar fashion to DNA and acting as an enzyme in a similar fashion to proteins. Our molcules are similar to the RNA-world model, since there is no distinction between genotype and phenotype - the machine (phenotype) is also the blueprint (genotype) for making copies of itself. In a reaction, each replicase molecule can act as either "copied" or "copier" when it encounters another molecule. The function of an RNA molecule is determined by its sequence. Regions of the sequence act as binding sites for other molecules, and other regions determine the reaction that the molecule might be involved in. Thus our model differs to modern biology, in which DNA and RNA sequences are composed of different subunits (nucleotides) from proteins (amino acids). Here, we want to explore how a sequence of codes can capture a desired function within a larger biological system. We do not currently concern ourselves with the translation of information between nucleotides and amino acids.

2.3 Inexact subsequence alignment

Copying is a bi-molecular process in which one partner acts as the template and one acts as the copier. In order to be copied, a replicase molecule must possess a sequence that matches a sequence on the partner replicase, and thus facilitate *binding* to it. Our earlier work [8] shows that the *rate* of binding is important in tuning a gene regulatory network. Our goal is to derive the bind probability from some similarity score between sequences. We do not wish to develop new algorithms to measure this similarity. Fortunately, a suitable subsequence alignment procedure is available to us. Smith-Waterman alignment [9] is the basis of comparison of genetic sequences to determine phylogeny between species. It is designed to detect non-random sequence matches and give a score to any alignment found. Gaps and mismatches can be accommodated for, and the penalty for each can be tuned as desired. This algorithm forms the basis of our molecule binding procedure, here applied to complementary alignments.

2.4 Protein folding and stack-based programming languages

Biological genome maintenance is carried out by proteins (folded sequences of amino acids) and small RNA molecules (sRNAs), both of which are built following a specification on the genome. The functional characteristics of the protein arise from its folded shape, which depends on the chemical properties of the amino acids. The sequence of amino acids forms the primary (1°) structure of the protein, but the process of folding into the tertiary (3°) structure is not explicitly encoded on the DNA, emerging from the chemical properties of the amino acids themselves. The functionality of the protein is thus a product of the DNA-based specification of the amino-acid sequence, the physical and chemical properties of the amino acids, and the physical and chemical properties of the protein structure itself. There is a high degree of interdependency between these factors. Proteins' folded shapes are highly structured and often flexible. Reactions occurring between proteins and other molecular entities cause changes in the tertiary structure of the protein,

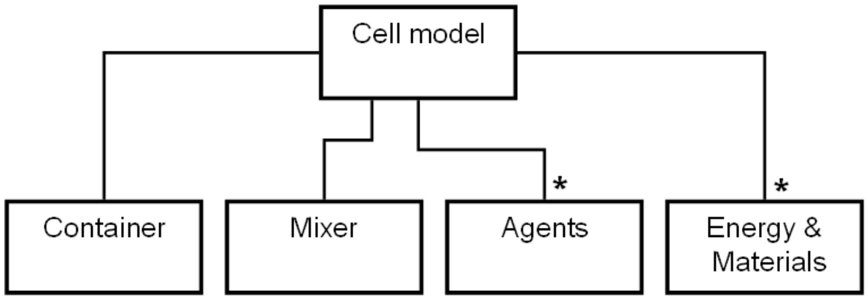


Figure 1: Components of a simple cell model

which in turn change the reactive properties of the protein. Sometimes the shape of the protein is the most significant factor. For example, proteins can form loops that can clamp onto DNA, physically obstructing other proteins from interacting with it. At other times, the charge on the protein's surface changes the conformation of its neighbours. Over evolutionary time scales, the genetic blueprint develops a deep (but implicit) interaction with these properties.

Protein folding (and indeed RNA folding) are difficult processes to emulate (beyond the current state of the art [10]). However, faithful rendering of molecular shape is not explicitly required for the binding scheme we are developing - we simply require some inexact matching process that emulates the effect that "docking" has on the chance of molecules coming together to react. If we abandon a shape-based representation, we will also require a means of deriving the function of a molecule from its sequence alone without some shape-based process. Fortunately, there is pre-existing work in this area also, in the form of stack-based programming languages where folding is represented abstractly by the concept of program flow. There is a rich history of ALife based on the concept of individual-as-program. Key examples are Tierra [7], AVIDA [11], nanopond [12] and "BF" [13]. All these place heavy emphasis on the genotype: each "organism" has a template code, plus some registers and some energy, which together confer function and fitness on the individual. The phenotype is forced to be simple: essentially a sequence of instructions, plus a miniature processing architecture that executes the genetic instructions. In these systems the processing architecture of each organism is not subject to heritable change. We argue that the ALife organism needs to be richer. The gene sequence must encode as much of the function of the organism as possible. As a first step towards this goal, we use executing sequences as the basis for the biochemistry of the organism, rather than as a representation of the entire organism in one unit. By stating that an organism is composed of a *set* of simple executing sub-programs, we can make simpler sub-units, draw a closer analogy with functioning proteins, and make more of the processing machinery available to evolve.

2.5 Summary

We have listed above the biological phenomena that have led us to the design of our molecular model. This model consists of a space in which computational agents mix, bind and react with each other. The mixing is an external process, but the binding rate, and the nature of the reaction are encoded on the sequence of instructions that define each molecular species. Having given background to the processes that have led us to develop this model, we now turn to the description of the model itself.

Reaction	Rule format		
Binding:	Agent + Agent	→	Reaction
Production:	Reaction	→	Reaction + Agent
Dissociation:	Reaction	→	Agent + Agent
Decay:	Agent	→	Ø

Table 1: The four types of reaction rule in the metabolic controller

3 Cell-level architecture

This section enlarges upon the description of the domain model in [8]. The metabolic model is composed of four components, as shown in figure 1.

Firstly, there exists a *container*, which specifies the volume and dimensionality of the space in which the agents exist. We specify a simple 2D container of area $v_c = 2500$ units.

Secondly, we have a set of metabolite *agents*, of area $v_a = 10$ units which are present in varying quantities in the container, analogous to the various quantities of different molecular species in a biological system. Agents are considered to be *bound* or *unbound*. If an agent is bound to another agent, then one member of the pair is executing its program (it is *active*), whereas the other member of the pair is *passive*. The program that the pairing executes is encoded in the sequences of the bound agent pair. We refer to this as the *Reaction*.

Thirdly we have a stochastic *mixer*, which governs the movement and changes in adjacency of the elements within the container. For a bimolecular reaction such as the bind B, our mixer utilises a simple propensity function $\mathrm{P}(B)$, which estimates the probability of two agents being sufficiently close enough for the reaction to occur. For any one agent in a bimolecular reaction, the chance of the second agent in the reaction being close enough to react is:

$$\mathrm{P}(B|v_c, v_a, n) = 1 - (1 - (v_a/v_c))^n \tag{1}$$

where n is the number of instances of the second agent in the metabolism. Space in the system is represented abstractly via the ratio of container area to agent area. Apart from this consideration, the model is aspatial.

Fourthly, energy and raw materials are available to the agents as they react. These are minor components to the model. Energy is supplied to the system at a fixed rate per time step. The only raw materials in the system are the symbols used in construction of the string. These are assumed to be present at saturation throughout the experiment, although conservation of mass could be implemented in the system straightforwardly. The state of the system is recorded in a series of discrete time steps. Each timestep is processed via examination of all agents in a random sequence. One binding attempt or one execution of an instruction requires one unit of energy. Energy not used in one time step is available in the next. Energy is modelled as a continuum since the computational cost of representing energy via individual particles is too great. This is an appropriate abstraction for the problem domain, and is in line with current practice in systems biology [14]. Waste materials, and materials required for construction of agents are assumed to be approximately constant and are not represented in the current model.

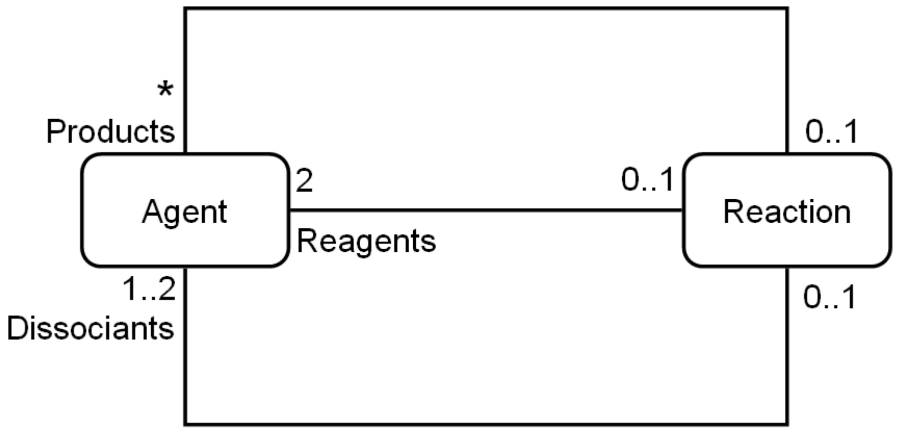


Figure 2: Class diagram for Agents and Reactions.

4 Reactions between agents

In earlier work [1] , we have shown that a rich regulatory system can be built following reactions with no more than two reactants and no more than two products. There, we specified the reactions in a metabolism via a simple reaction table. Having shown that a rich system can be created when using a simple set of reaction rules, we can now extend the model to make the system evolvable. Our replicase system is built from reactions in the form shown in table 1. There are four events in the system which change the agents making up the metabolism at any one time. Each of these events is controlled by the sequence of codes belonging to the agents, under a stochastic timing process.

Binding occurs when two reactants combine to create a single product. Binding is the only bimolecular reaction permitted. Bimolecular reaction rates are governed by the concentration of the two reactants in the system (via equation 1) and a further reaction rate specified by the alignments of the molecular strings as described below. In our test model, the two agents that combine to form a bind are both instances of the replicase agent. Once agents are bound, they form a *reaction*. It is helpful to consider a reaction as a distinct computational entity. Whilst agents are in a reaction, the sequence of symbols acts as a program, making use of other structures belonging to the agent in order to carry out its function.

Production is the creation of a new agent during a reaction. Production of a new molecule is encoded as a special instruction in the sequence of one of the agents, which must be suitably positioned on the sequence in order for a new molecule to be created. The structure of the new agent is determined by the microprogram of the reaction.

Dissociation is the splitting of a reaction into its two constituent agents. This occurs after the microprogram of the reaction has terminated. The sequence of any agent may be changed by the reaction.

Decay occurs only to agents that are not in a reaction. Decay is an essential component of our replicase model, since it allows an equilibrium state to arrive where the number of replicase molecules is roughly constant. For computational simplicity, decay is spontaneous - agents are instantaneously deleted from the system with none of the degradation of structure that can occur in biological systems. Decay is stochastic, but the probability of decay is a function of the length of the sequence. This is a crude way of ensuring that molecules thet are long, and so expensive to build tend to persist in the metabolome, without having molecule fragments floating around and

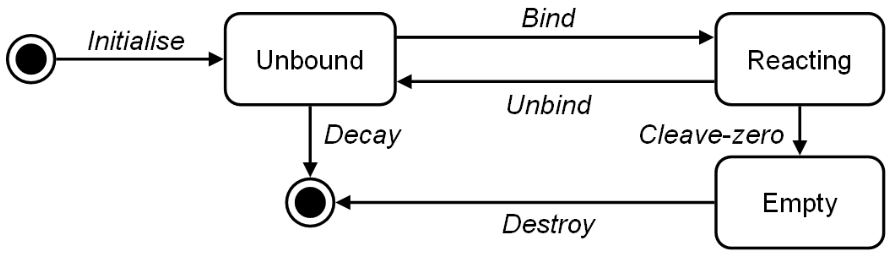


Figure 3: State chart for Agents

reacting with other, complete molecules in the system, although we recognise that this is not found in biology. We use a decay probability proportional to $1/l^2$, where l is the length of the string. Note also that this is "passive" decay. There is scope to build a "destructase" molecule, whose microprogram would chop up anything that bound to it into smaller molecules. The resulting fragments would then be more likely to decay, since they are shorter.

It is important to note that the rates in each of the event types above are controlled by the sub-symbolic representation of the molecular agent. We specify binding rate B via an alignment of sequences. The dissociation rate is controlled by the number of time steps it takes the program specified by the sequence to execute plus a stochastic term based on availability of energy. An instruction will only be executed if energy is available. Decay is governed by the length of the string.

The relations between the objects and events in our system are illustrated in figures 2 to 4. Figure 2 is a class diagram of the two object types in our system, namely *agent* and *reaction*. An agent can participate in a reaction in three ways. Two *reagents* bind to form the reaction complex. The reaction can produce new agents in the form of *products*. Any reagents remaining when the reaction terminates are *dissociants* (there can be one or two of these).

A state chart for the agent class is shown in figure 3. The bind, unbind (i.e. dissociate) and decay events are all instantaneous. Unbound agents are essentially inert until a bind event or a decay event happens. When two agents bind, they are said to be *reacting*. During a Reaction, new agents can be produced. When the reaction ends, the partners in the reaction revert to being inert agents. An agent's string specifies its binding properties. Once in a reaction, the same string becomes the program. There are no delimiters between the binding region and the program region. A reactant can be destroyed whilst in a reaction if a cleave operation (described below) occurs at the first symbol, producing a molecule with no sequence (shown as "cleave-zero" in figure 3.

An activity diagram for objects in the system is shown in figure 4. It illustrates how the two agents form a reaction, and how new agents may be produced in that reaction.

5 Designing an artificial replicase

To test the concept of combining an artificial chemistry with a stack-based program specification, we set ourselves the goal of designing a self-maintaining chemical system constructed of multiple instances of a single molecular species. We call this molecule a "replicase", since it must create copies of itself to maintain the population. To do this, an instance of the molecule must bind to another instance of the molecule, and through some process create a third instance of the molecule. Following this, the original two molecules must dissociate. The molecules are also subject to a

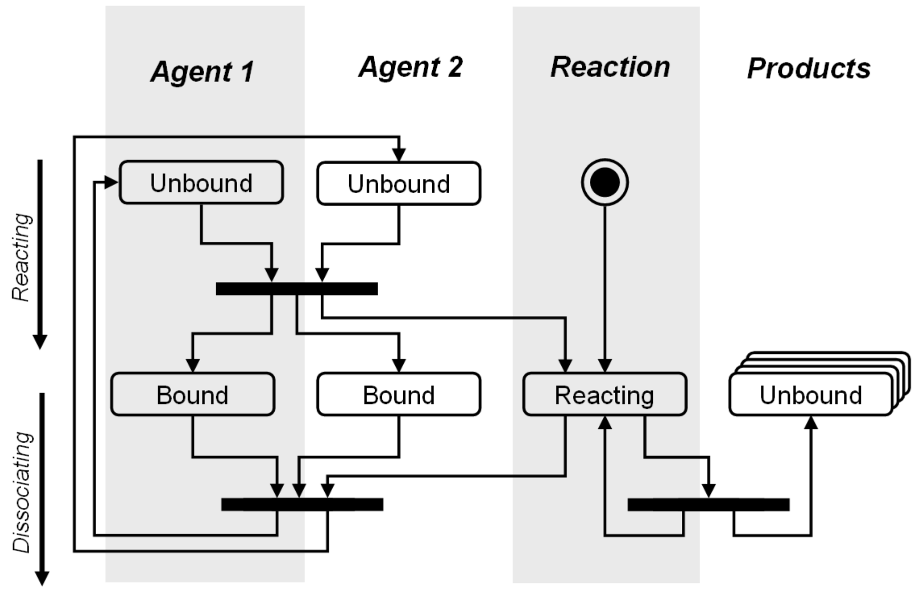


Figure 4: Activity diagram for Agents and Reactions.

decay process, such that they effectively have a "lifetime".

Our molecular microprograms are analogous to proteins and sRNAs, where sequences of amino acids fold to form molecular machines. Our molecules have no shape or explicit dimension. The uniqueness of a molecule is encoded in its sequence of symbols - its string. We have therefore acquired the habit of calling our molecules *Stringmols*. We use program control flow as an analogy of the tertiary structure of a protein for a sequence of program symbols, since program flow is likely to change dramatically with small changes in the symbol sequence, just as the shape of a protein can change via mutation of a single amino acid. We place heavier emphasis on the process of molecular binding than do AVIDA and variants. In functioning microprograms, sequences describe one or more binding regions and when bound, a microprogram is executed from the sequence of instructions commencing at the start of the bind. Thus a binding event triggers a reaction sequence, which is encoded on the molecule and run as a program. Different programs can be encoded on the same molecule, and be triggered by binding at different sites.

The sequence of the designed replicase is shown in figure 5. There are three distinct regions. The first controls the binding probability for two agents. Region 1 of one replicase molecule binds to region 2 of another replicase molecule by a process of complementary binding as described below. The centre of the sequence is "junk". This region carries out no function, but its presence adds to the length of the sequence and is thus important in determining the decay rate of the molecule. The final region specifies the reaction that the replicase performs. It is here that the sequence can be interpreted as a program (described below). Note that the three regions of the molecule are not delimited in any way. There is thus scope to design a molecular sequence which has more than one binding site and/or program. A sequence could be interpreted as all junk if it doesn't bind. A sequence can be very short and simply bind to another molecule type to prevent it reacting with anything else. We believe that this flexibility is one of the strengths of the system.

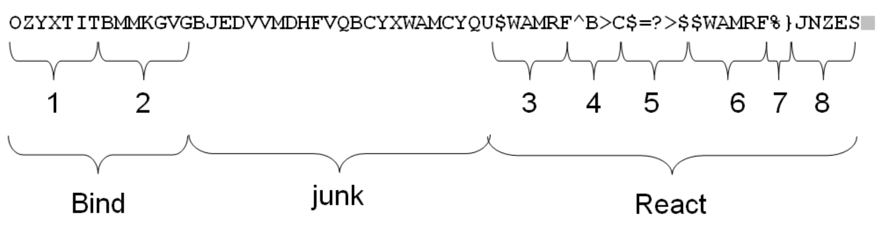

OZYXTITBMMKGVGBJEDVVMDHFVQBCYXWAMCYQU$WAMRF^B>C$=?>$$WAMRF%}JNZES
1
2
3
4
5
6
7
8
Bind
junk
React

Figure 5: Sequence of an artificial replicase

6 Binding using inexact subsequence alignment

We require a model of biochemical binding that is simple to implement. The main idea is that when two molecules are brought together, there is a *probability* of binding - the two molecules will join together with a particular probability, or they will not join. There is no intermediate state in our abstraction, simply a before and after state. The probability of biological molecules binding is due to many things (temperature, pressure, pH), but the process that we emulate here is that the shape and composition of the two molecules is the key evolved factor in determining whether they join together. Thus, our binding scheme is concerned with some comparison of the strings of the two agents in a bind. In this section, we consider how to build a binding scheme which uses inexact sequence alignments as the basis of the probability of two agents binding. We only discuss strings here - the molecular apparatus needed to run the sequence as a program is described in the sections that follow.

Although subsequence alignments offer a method for obtaining a score for an alignment, several design decisions must be made in order to use the approach to simulate binding between molecules. An *alphabet* of symbols must be chosen. We need to ensure that two instances of the same species do not always bind strongly to each other. This can be achieved by specifying that binding occurs via a process of *complementary alignment*. We must then decide the complementary symbol of each symbol in the alphabet. An appropriate *scoring scheme* for the alignments must be available, and an appropriate function to convert the score to a *binding probability* must be developed. This section describes our design choices. In addition, section 6.4 describes the Smith-Waterman algorithm (SWa), which forms the basis of our approach. We have not changed the algorithm in any way, merely developed procedures to construct input and output functions that make it suitable for the task at hand.

6.1 Sequences of symbols

Strings are composed of sequences of symbols from a finite alphabet Σ. Symbols are of two types. Firstly, we have T, the set of twenty-six *template codes* from the uppercase alphabet:

$$T = \left\{ \begin{array}{l} \texttt{`A'}, \texttt{`B'}, \texttt{`C'}, \texttt{`D'}, \texttt{`E'}, \texttt{`F'}, \texttt{`G'}, \texttt{`H'}, \texttt{`I'}, \texttt{`J'}, \texttt{`K'}, \texttt{`L'}, \texttt{`M'}, \\ \texttt{`N'}, \texttt{`O'}, \texttt{`P'}, \texttt{`Q'}, \texttt{`R'}, \texttt{`S'}, \texttt{`T'}, \texttt{`U'}, \texttt{`V'}, \texttt{`W'}, \texttt{`X'}, \texttt{`Y'}, \texttt{`Z'} \end{array} \right\} \quad (2)$$

which are used for alignment of strings and for program flow control only. We use the complete upper case alphabet to facilitate rich sets of template sequences in our molecular structures, and to allow mnemonic sequences to be used whilst initialising our molecules. Secondly, we have Φ, the set of seven *function codes*, which are used for the seven different operations in our micropro-

Σ	**ABC$DEF>GHIJ^KLM=NOP?QRS}TUV%WXYZ**
$C(\Sigma)$	**NOP$QRS>TUVW^XYZ=ABC?DEF}GHI%JKLM**

Table 2: The set of symbols Σ , and its complement $C(\Sigma)$.

gramming language:

$$\Phi = \{ \text{`\$'}, \text{`>'}, \text{`\^{}'}, \text{`?'}, \text{`='}, \text{`\%'}, \text{`\}'} \} \tag{3}$$

The alphabet Σ is the union of the set of template codes T and the set of function codes Φ:

$$\Sigma = T \cup \Phi \tag{4}$$

Symbols are always written in **THIS** font in our notation.

Each molecule has a program, encoded as a string s, consisting of a sequence of codes from the alphabet. s is a member of the set of all possible strings Σ^*:

$$s \in \Sigma^*. \tag{5}$$

A symbol with index i on string s is indicated by

$$s_i, \quad i \in 0...(m-1) \tag{6}$$

where m is the length of the string.

An important property of these strings of symbol sequences is that they contains subsequences that may be aligned with other, non-contiguous subsequences

6.2 Complementary string alignments

We use an adapted form of complementary base pairing as part of our alignment strategy. The complement of a string s is the string $\mathrm{C}(s)$. The complement of all symbols in Σ is shown in table 2. Our (untested) intuition is that template symbol complements could be chosen arbitrarily, but that functional symbol complements offer a potential hazard — they may force long functional regions to act as templates, and thus suppress the generation of alternative binding schemes. In these early stages of designing this chemistry, we have specified the complement of a template symbol to be 13 letters downstream in the English alphabet and that functional symbols act as complements of themselves. Note also that the *order* of the symbols is important, as it is used to calculate mismatch scores in the alignment process - the nearer a symbol is in the list, the less severe the mismatch.

The process of alignment is shown in figure 6. The sequence '**OZYXTIT**' from region 1 of string 2 is bound to the sequence '**BMMKGVG**' from region 2 of string 1, since the complement of region 2 forms an alignment with region 1. The complementary nature of the alignment is necessary to prevent copies of strings binding to each other perfectly, and to allow more than one instance of a sequence on a string to align to a single complementary sequence. As we shall see, this factor also allows subroutines ("goto" statements) to be developed, which the microprograms can exploit whilst functionality is evolving.

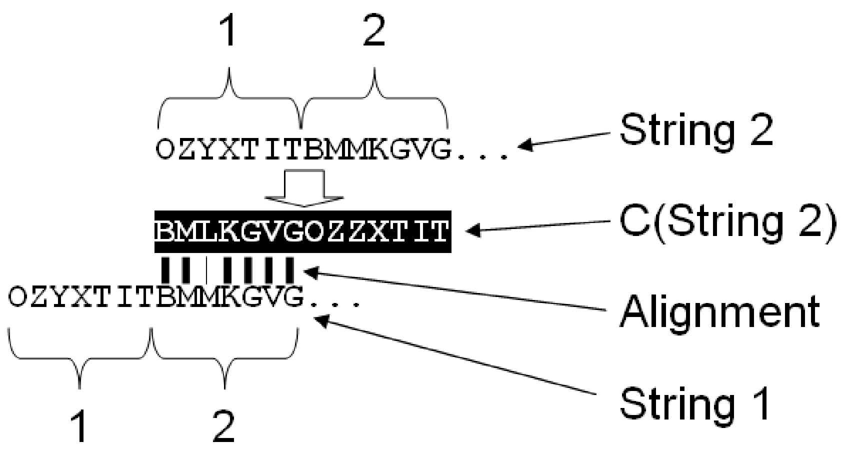


Figure 6: Complementary matching to form a bind. The sequence '**`OZXYTIT`**' has a complement (shown in white lettering) which aligns with '**`BMMKGVG`**'. The alignment is not perfect as the third letter in the sequence forms a mismatch.

6.3 A substitution matrix for microcodes

Alignments are considered by comparing strings on a symbol-by-symbol basis. A *score* must be available for a match, a mismatch, or an indel (an insertion or deletion of a symbol). When attempting to find the best alignment between string s with index i and length m, and string s' with index j and length n, it is desirable to have a scoring system that reflects the chance of a mutation (or indel) from the symbol at s_i to the symbol at s'_j.

In studies of biological sequences, the chance of mutation is estimated by reference to large numbers of studied sequence phylogenies. When comparing two sequences of symbols s and s', three cases need to be assigned a score: a match from one symbol to another; a mutation from one symbol to another; and a gap in the alignment caused by the insertion of a symbol on one string or a deletion on the other (an *indel*). These values are stored in a substitution matrix, ω. This matrix is of size $(n(\Sigma) + 1) \times n(\Sigma)$, where $n(\Sigma)$ is the number of symbols in the alphabet. The extra row is used to store the indel score for a particular symbol and has index $-$. The function $\overrightarrow{\Sigma}(s_i)$ returns the index on ω of the symbol at s_i.

For sequences from biological systems the evidence for mutation rates that have occurred historically is used to give a score to the differences in the sequences under consideration. Even though our model has no mutation at present, we still require a substitution matrix to obtain an alignment of strings via SWa. In our application, we must generate substitution values from scratch in the absence of a historical record of mutations.

Unconstrained by the physical world, we are free to define the substitutions between symbols in any way we choose. As a first step, we have specified that codes mutate forward or backward through a sequential loop. The functional symbols are intermingled with the template symbols throughout the loop to allow functional changes for any symbol to be always only a few mutations away and reduce any clustering of functional codes to form "functional hotspots" in mutation sequences. Substitution mutations can be forward or backward in the sequence. Thus the symbol '**`D`**' can mutate to one or other of its neighbours: '**`$`**' or '**`E`**', and at the start of the loop the symbol '**`A`**' can mutate to '**`B`**' or (going to the end of the sequence) '**`Z`**'. (See table 2 for the full sequence.) A change from '**`C`**' to '**`J`**' would require at least seven mutations (if there happen to

	A	**B**	**C**	**$**	**D**	**E**	**F**	**>**	**G**	**H**
A	1.00	−0.12	−0.25	−0.38	−0.50	−0.62	−0.75	−0.88	−1.00	−1.12
B	−0.12	1.00	−0.12	−0.25	−0.38	−0.50	−0.62	−0.75	−0.88	−1.00
C	−0.25	−0.12	1.00	−0.12	−0.25	−0.38	−0.50	−0.62	−0.75	−0.88
$	−0.38	−0.25	−0.12	0.50	−0.12	−0.25	−0.38	−0.50	−0.62	−0.75
D	−0.50	−0.38	−0.25	−0.12	1.00	−0.12	−0.25	−0.38	−0.50	−0.62
E	−0.62	−0.50	−0.38	−0.25	−0.12	1.00	−0.12	−0.25	−0.38	−0.50
F	−0.75	−0.62	−0.50	−0.38	−0.25	−0.12	1.00	−0.12	−0.25	−0.38
>	−0.88	−0.75	−0.62	−0.50	−0.38	−0.25	−0.12	0.50	−0.12	−0.25
G	−1.00	−0.88	−0.75	−0.62	−0.50	−0.38	−0.25	−0.12	1.00	−0.12
H	−1.12	−1.00	−0.88	−0.75	−0.62	−0.50	−0.38	−0.25	−0.12	1.00
-	−1.33	−1.33	−1.33	−1.33	−1.33	−1.33	−1.33	−1.33	−1.33	−1.33

Table 3: Sample values of the substitution matrix ω. Indel scores are presented on the bottom row of the table

be no anti-clockwise mutations), and '**L**' to '**S**' would require at least eight mutations.

It is straightforward to implement this mutation scheme in an evolving system. Fully random mutations of fixed probability between all codes would create different substitution values. It is difficult to know *a priori* what weight a scheme such as this will have on the efficacy of an evolutionary algorithm. The potential benefits are that codes with closely-related properties can be given "cheap" substitution scores, and thus allow smoother exploration of the gene-space. In this current contribution, we can only speculate about the effects of this approach.

Although this strategy allows us to give *relative* values of substitution scores, we need to fix the values in absolute terms in order to create the substitution matrix. Smith and Waterman [9] state that for a four-letter alphabet, a substitution weight of $-1/3$, a match weight of 1 and an indel weight of $-4/3$ gives an average score of zero for randomly-generated strings. Shpaer [15] reports a substitution matrix for amino acids. This gives similar average substitution weights if the values are normalised for the average match score.

Our substitution matrix ω is filled as follows. Let $a = \vec{\Sigma}(s_i)$ and $b = \vec{\Sigma}(s'_j)$. The diagonal values of the matrix, representing the score if symbol a is found at s_i and s'_j, are set to 1.0 for template codes, and 0.5 for function codes:

$$\omega(a,a) = \begin{cases} 1.0 & \text{if } a \in T \\ 0.5 & \text{if } a \in \Phi \end{cases} \tag{7}$$

This places appropriate emphasis on the binding and executional function of the two symbol types. These scores occupy the leading diagonal of ω as shown in table 3. Scores for mismatches are based on the distance between symbols in the alphabet, and are calculated via the *substitution equation*:

$$\omega(a,b) = \frac{-\min(|a-b|, N-|a-b|)}{\sum_{k=1}^{N} \min(k, N-k)/N} \qquad \text{if } (a \neq b) \tag{8}$$

where the size of the alphabet $N = n(\Sigma)$. This sets the average score for substitution to -1. This average penalty is more severe than Smith and Waterman recommend, and has the effect that more alignments will score zero (see below). We have adopted this strategy because our application currently utilises alignments that are significantly shorter than the biological sequence alignments that SWa was designed for. A more severe penalty on mismatches helps to ensure that accidental alignments occur at an appropriate rate in our model. Note also that substitution scores depend entirely on the sequence in which symbols from Σ are placed in ω. Scores for indels are fixed to a constant:

$$\omega(-,a) = -4/3 \tag{9}$$

These indel values are shown on the bottom row of table 3. The penalty for substitution gets more severe with distance from the leading diagonal, indicating that a *sequence* of mutations would be required for distant symbols to be substituted for one another.

6.4 Calculating the alignment score using the Smith-Waterman algorithm

With our substitution matrix to hand, we can calculate the highest-scoring alignment for any pair of strings formed from Σ using the Smith-Waterman (SWa) algorithm. For two sequences s and s' of length m and n respectively, the algorithm populates a matrix H of size $m+1$ by $n+1$ with the similarity scores between all symbols in two sequences s and $C(s')$. Each cell in the matrix H is initialised to zero. The first row and the first column in H act as indicators for the start of the strings. The presence of these extra cells allows processing to be done in a uniform manner. The algorithm starts at cell $H(i=1, j=1)$ (assuming zero-indexing), and proceeds row by row until cell $H(i=m, j=n)$ is reached. The value of each cell in this submatrix is [16]:

$$H(i,j) = \max \begin{cases} 0 & \text{No alignment} \\ H(i-1,j-1) + \omega(a,b) & \text{Match/Mismatch} \\ H(i-1,j) + \omega(-,a) & \text{Deletion} \\ H(i,j-1) + \omega(-,b) & \text{Insertion} \end{cases} \tag{10}$$

Where $w(x,y)$ returns the appropriate value from the substitution matrix at x,y and $w(-,x)$ indicates the row containing the indel value for symbol x. Scores for sequences of pure matches and mismatches run parallel to the leading diagonal of H, whereas insertions and deletions cause a horizontal or vertical shift in the path of the maximal score. In addition to H, a *trace matrix* T is produced, which indicates whether the value in each cell is the result of a match/mismatch, insertion or deletion. A illustrates this process with an example.

Once processed, the maximum local alignment $\phi_{S,S'}$ can be found by detecting the highest value $H(i_\beta, j_\beta)$ in the matrix H, and then tracing the alignment back to its origin using T. The origin of the alignment is the cell $H(i_\alpha, j_\alpha)$, in which all upper-left neighbours are zero-valued is obtained. The alignment site on s runs from s_{i_α} to s_{i_β} and the alignment on s' runs from s'_{j_α} to s'_{j_β}.

Let the index on s and s' of the zero-scoring cell in H which marks the start of the alignment be s_α and s'_α and the index of the highest-scoring cell in H which marks the end of the alignment be s_β and s'_β. The length λ of the bind is the minimum length of the alignment on the two strings:

$$\lambda = \min(s_\beta - s_\alpha, s'_\beta - s'_\alpha) \tag{11}$$

Note that the minimum alignment length can never be less than the alignment score. Alternative schemes might make use of the maximum length alignment.

Given the ingredients above, we have implemented a function ϕ, which takes as arguments two strings, s and s' and a substitution matrix ω. The function finds the Smith-Waterman alignment between s and $\mathrm{C}(s')$, and returns the score σ and length λ of the alignment:

$$\sigma_{s,s'}, \lambda_{s,s'} = \phi(s, s', \omega) \tag{12}$$

6.5 From alignment scores to bind probabilities

The Smith-Waterman algorithm was designed to give a score to subsequence alignments for the analysis of phylogeny. It was not intended for use as a metric for binding. In order to use SWa for binding, we must define a function which maps the alignment score to a probability of bind success. Note also that bind *success* is distinct from bind *strength*. The strength of a bind, once formed is uniform and absolute. The bind can only be broken by terminating the program that is run when the bind is formed.

In order to calculate the alignment probability for any two strings s and s', we first obtain $\phi(s, s', \omega)$, and use the score σ and the length λ to define the binding site and derive a probability of binding, $\mathrm{P}(\phi(s, s', \omega))$. Longer subsequences tend to score more highly. We can take these values and derive a binding probability via reference to the maximum possible score for a match of a given length.

For an alignment of score σ, we define the binding probability P as:

$$\mathrm{P}(\phi(s, s', \omega)) = (\sigma/\lambda)^{\lambda} \tag{13}$$

This gives us the range of binding strengths we need to build metabolic networks of computation entities. If a particularly weak bind is sought, then a longer subsequence alignment with sufficient mismatches can deliver that weak bind. Note that raising the fraction σ/λ to the power of λ means that low-scoring alignments have very weak binding probabilities. We have found that this approach makes it easier to initialise a system by hand, since fewer unplanned significant alignments emerge during the design of the molecular species for our experiments. We plan to experiment with ways of obtaining the most useful (evolvable) range of binding probabilities in future work.

We have now shown how to obtain a bind from a string alignment. When two agents bind successfully, they form a reaction. Before we describe how reactions are run as microprograms as specified by the agent sequences, we must detail the apparatus that is needed to carry out the reaction.

7 Molecular structure

In order to use the string sequences to drive a range of intermolecular reactions and thus develop new molecules with their own sequences, we require some means of using the symbol sequence to specify manipulation of the strings. In this section, we describe the structures that we use to allow molecules to run string microprograms. Recall our biochemical analogy: two adjacent molecules binding to each other, initiating a reaction that creates products. The characteristics of all of these events are encoded on the string of the molecular analogues. In addition, each molecule has an architecture consisting of a set of pointers and flags that govern the execution of the program. The key concept here is that a bound pair of molecules forms a complete computational entity. The sequences of both molecules are used in the program. Pointers exist in pairs, with only one of the pair ever "active" at any one time. The "active" pointers belonging to a single molecule in the bound pair control program flow and read/write operations, and are similar to the pointers used in the AVIDA system [11]. A record of which pointer is active in each pair is stored in "activator" flags. Finally, the state of the molecule is recorded in a "status" flag.

In this section, we denote a molecule as $\mathcal{M}$, and use the operator ':' to denote structures belonging to the molecule. For example the string s^1 of a molecule $\mathcal{M}^B$ is referred to as $\mathcal{M}^B : s^1$. We use the notation $\perp$ to indicate where structures are undefined. Each molecule $\mathcal{M}$ has access to a string s^1, a string from a partner molecule s^2, two sets of pointers p^1 and p^2, a set of activators a, and a status flag δ:

$$\mathcal{M}\begin{cases} s^1, s^2 & \in \Sigma^* \cup \perp \\ p^1 & \in \mathbb{N}^4 \\ p^2 & \in \mathbb{N}^4 \cup \perp \\ a & \in \{1,2\}^4 \\ \delta & \in \{\mathrm{UNB};\mathrm{ACT};\mathrm{PAS}\} \end{cases} \tag{14}$$

7.1 Molecular states

Our computational molecules exist in one of three states δ: unbound (UNB); active (ACT); passive (PAS):

$$\delta = \{\mathrm{UNB};\mathrm{ACT};\mathrm{PAS}\} \tag{15}$$

These states govern the availability of the other structures that comprise the molecule as a reaction proceeds.

When two molecules bind, one molecule takes an active state and one takes a passive state. We adopt the convention that the active molecule has index 1 and the passive molecule has index 2, reflecting the "grammatical person" in the English language (the first person is "I" and the second person is "you"). The molecule in the active state (δ = ACT) runs the reaction initiated by the bind.

7.2 Molecular strings

The *sequence* s^1 of a molecule defines the molecule's binding and program. The reaction between two molecules is encoded on the strings of the two molecules. We use the notation $\mathcal{M} : s^2$ to reference the string of the partner of a bound molecule. The definition of a sequence is shown in equation 5.

To illustrate this concept, consider two bound molecules, with sequences s^A and s^B. The active molecule $\mathcal{M}^A$ has $\mathcal{M}^A : s^1 = s^A$; $\mathcal{M}^A : s^2 = s^B$; $\mathcal{M}^A : \delta = \mathrm{ACT}$. The passive molecule $\mathcal{M}^B$ has $\mathcal{M}^B : s^1 = s^B$; $\mathcal{M}^B : s^2 = s^A$; $\mathcal{M}^B : \delta = \mathrm{PAS}$. By contrast, an unbound molecule $\mathcal{M}^U$ with sequence s^U has $\mathcal{M}^U : s^1 = s^U$; $\mathcal{M} : s^2 = \perp$; $\mathcal{M}^U : \delta = \mathrm{UNB}$.

7.3 Pointers

Programs are executed when molecules bind, via manipulation of pointers and the symbols that they point to. There are four pointer types: instruction $p_{\mathbf{I}}$; flow $p_{\mathbf{F}}$; read $p_{\mathbf{R}}$; write $p_{\mathbf{W}}$. Pointers take the value of the index of the symbol on a molecule's string s to which they are pointing. The set of four pointers is:

$$p = \langle p_{\mathbf{I}}, p_{\mathbf{F}}, p_{\mathbf{R}}, p_{\mathbf{W}} \rangle \in \mathbb{N}^4 \tag{16}$$

Each molecule has two sets of the four pointers. One set, p^1 is used to interact with the string of the molecule itself, the other, p^2, is used to interact with the string of a partner molecule in the reaction. Only one instance of each pointer type is ever available for interaction at any one time. Pointers are only set to a defined value whilst the molecule is active (i.e. $\mathcal{M} : \delta = \mathrm{ACT}$).

7.4 Activators

Activators specify which of the two instances of each pointer type is currently active. Activators take values:

$$a = \langle a_{\mathbf{I}}, a_{\mathbf{F}}, a_{\mathbf{R}}, a_{\mathbf{W}} \rangle \in \{1, 2\}^4 \tag{17}$$

Each member of $\mathcal{M} : a$ takes one of two values 1 and 2, indicating which pointer set $\mathcal{M}{:}p^1$ or $\mathcal{M}{:}p^2$, and thus which molecule's string $\mathcal{M}{:}s^1$ or $\mathcal{M}{:}s^2$, that a particular pointer type is currently acting upon. All activators are set to 1 when a molecule's state becomes active, so placing all the pointer types on the active molecule.

7.4.1 Binding molecules

When molecules are created, their state is unbound: $\mathcal{M} : \delta = \mathrm{UNB}$. They are not in the process of reacting, and so no pointer manipulation is taking place. Precisely two molecules are involved in a binding event. Within this bind, one of the molecules is considered to be executing its program, much as a biological enzyme can be thought to be acting upon a substrate. By convention, we specify that molecule $\mathcal{M}^1$ is the *active molecule* the molecule which is executing the program, with state $\mathcal{M}^1 : \delta = \mathrm{ACT}$ and molecule $\mathcal{M}^2$ is the *passive molecule* so $\mathcal{M}^2 : \delta = \mathrm{PAS}$. When two molecules $\mathcal{M}^A$ and $\mathcal{M}^B$ are brought together, we must first test whether they bind, and if they do bind, we must determine which becomes the active molecule $\mathcal{M}^1$ and which becomes the passive molecule $\mathcal{M}^2$.

To determine whether the molecules bind, an alignment test is carried out to obtain the binding probability $\mathrm{P}(B) = \mathrm{P}(\phi(\mathcal{M}^A : s^1, \mathcal{M}^B : s^1, \omega))$ and arrange the apparatus of each molecule accordingly. If a bind is successful, a handshaking process is required to determine at which symbol the execution of the reaction starts. For two sequences s and s' the executing sequence s^1 is that for which the start of the binding site s_α is furthest from the start of the sequence. To determine whether a bind has occurred, $\mathrm{P}(B)$ is compared with a random number r drawn from the range

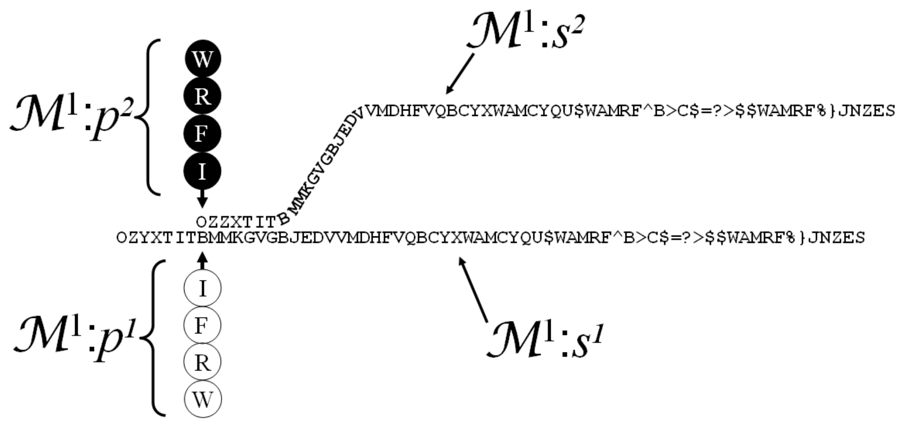

$\mathcal{M}^1:s^2$
$\mathcal{M}^1:p^2$
W
R
F
I
OZZXTITBMMKGVGBJEDVVMDHFVQBCYXWAMCYQU$WAMRF^B>C$=?>$$WAMRF%}JNZES
OZYXTITBMMKGVGBJEDVVMDHFVQBCYXWAMCYQU$WAMRF^B>C$=?>$$WAMRF%}JNZES
$\mathcal{M}^1:p^1$
I
F
R
W
$\mathcal{M}^1:s^1$

Figure 7: Initial state of a reaction between two replicase agents. Since $\mathcal{M}^1 : a := \{1\}^4$, all active pointers (shown as black text in white circles) are on $\mathcal{M}^1 : s^1$.

$[0, 1)$. The state of the two molecules changes as follows:

```
if($r <$ P$(B)$)
    if($\mathcal{M}^A : s^1_\alpha <= \mathcal{M}^B : s^1_\alpha$)
        $\mathcal{M}^1 := \mathcal{M}^A; \mathcal{M}^2 := \mathcal{M}^B$
    else
        $\mathcal{M}^1 := \mathcal{M}^B; \mathcal{M}^2 := \mathcal{M}^A$
    $\mathcal{M}^1 : \delta :=$ ACT$; \mathcal{M}^2 : \delta :=$ PAS                    (18)
    $\mathcal{M}^1 : s^2 := \mathcal{M}^2 : s^1$
    $\mathcal{M}^1 : p^1 := (\mathcal{M}^1 : s^1_\alpha)^4$
    $\mathcal{M}^1 : p^2 := (\mathcal{M}^1 : s^2_\alpha)^4$
    $\mathcal{M}^1 : a := (1)^4$
```

We would like the properties of the bind to act as determining factors for making this choice. Although there are many ways to decide which molecule is most appropriate to name as the executing sequence, we have gone for simplicity here, and use the distance of the start of bind to the start of each sequence to determine which sequence executes.

In our replicase example, this strategy makes scanning of the entire sequence of the passive molecule simpler if the complementary site is placed at the beginning of the sequence, since there is then no need to reposition the read pointer after binding. This is illustrated in figure 7. The active molecule's string s^1 is placed below the passive molecule's string s^2 in the figure. The active pointers are all placed at the start of the bind on s^1 (so $a^1 = 1$), and are shown in white with black text. The inactive pointers are all at the start of the bind on S^2, which happens to be the beginning of the string.

8 Overview of the Microcode library

We have implemented a limited set of *microcodes* - special symbols in the alphabet that allow us to perform computational analogues of molecular reactions. Table 4 provides a summary of these microcodes, which together manipulate the pointers and control the execution pathways of the molecular microprograms. We have carefully designed our codes such that they only have influence on the bound pair of strings. These instructions manipulate the positions of pointers,

Code(s)	Name	Default pointer	Description
A to **Z**	**n-op**		inactive template code and instruction modifier
$	**search**	***F**	shift ***F** to a matching template
>	**move**	***I**	shift pointer to the flow pointer
^	**toggle**	***I**	switch pointer to molecular bind partner
?	**if**	***I**	conditional single or double increment of instruction pointer
=	**copy**		insert at ***W** a copy of the symbol at ***R**
%	**cleave**		split a sequence and generate a new molecule
}	**end**		terminate execution and break the bond between the molecules

Table 4: Symbols and actions used in the current implementation of molecular microprograms.

and so control the execution pathways of the molecular microprograms. In the sections below, we indicate the symbols and names for our codes in bold courier font.

8.1 Modifiers and associated templates

Microcodes work by shifting the position of pointers and manipulating symbols and structres at the pointer positions. We would like to limit the number of different microcodes in the system, but make them as flexible as possible by allowing them to operate on different pointers in certain circumstances. This is implemented by specifying *modifiers* to operational codes. On execution of an operational code, the microcode immediately downstream is checked. If it is a modifying code, then the pointer to which the executing code applies is substituted according to the modification. Instructions that make use of pointers always have a *default pointer*. This is the pointer that is used if a modifier is not present.

In addition to manipulating the pointer that a code operates on, it is also necessary to have some mechanism that allows pointers to make a comparision between sequences on strings. The general mechanism for this is to define a sequence immediately downstream of the microcode that is the complement of a subsequence elsewhere on the string. A series of template codes downstream of a functional code is the *associated template* of the functional code. The uses of the associated template are described below.

Table 5 shows examples of modifiers and associated templates. The *subject pointer* is the pointer that is subject to manipulation. There is always a default subject pointer, but the subject pointer can be changed by reference to modifiers.

Note that only three modifiers are used: **A**, **B** and **C**, so only $3/34$ instructions are modifiers, meaning that on average, the default pointer would be used 85% of the time $(1 - (3/34))$. This is probably a conservative bias, and we may want to increase the number of modifiers in subsequent work - possibly using all 26 template codes as modifiers.

8.2 Notation

In the following descriptions it is convenient to simplify some of the notation above. Since the active molecule in the reaction has all the apparatus required to run that reaction, we drop the $\mathcal{M}$ prefix in this section. Most of our microcode operations refer only to pointers which are currently

Example	Modifiers	Associated Template
`?$$`	No modifiers	No associated template
`?A$`	The subject pointer becomes ***I**, the instruction pointer.	No associated template
`?B$`	The subject pointer becomes ***R**, the read pointer	No associated template
`?C$`	The subject pointer becomes ***W**, the write pointer	No associated template
`?BNZRY$`	No modifiers.	The associated template is **`BNZRY`**

Table 5: Examples of modifiers and associated templates

active. The activators a indicate which pointer is currently active. Since only active pointers are used for most operations, it is convenient to drop the a where only active pointers are used. For example, $s_{\mathbf{F}}^{a_{\mathbf{F}}}$ indicates the symbol with index $\mathbf{F}^{a_{\mathbf{F}}}$ on the string that the active flow pointer currently references. Where this is clear, we drop the $a_{\mathbf{F}}$ and use the notation $s_{\mathbf{F}}$ to make the notation easier to read.

9 Auxiliary functions

These functions act as the building-blocks for the functional microcodes. We felt that they were too simple for their function to be represented as a unique code since there would be too many codes required for a novel function to emerge.

9.1 len

This function returns the length of a string. Thus for a string of length 12, $\mathrm{len}(s) = 12$. If the function is applied to a pointer, the function returns the length of any associated template, which is the number of template codes downstream of the pointer until a function code or the end of the string is encountered. Thus if a pointer p indexes the start of the sequence **`$JAMES%^`**, $\mathrm{len}(p) = 5$.

9.2 rand

The $\mathrm{rand}\,()$ function returns a pseudorandom number in the range $[0, 1)$.

9.3 swap

Each molecule possesses two instances of each pointer type. The pointer that the functions act on is specified by the activators $\mathcal{M}^1 : a$. The swap function acts on $1, 2 \rightarrow 2, 1$. Thus in the case of the instruction pointer:

$$\begin{aligned} &\mathrm{swap}(x) : \\ &\quad \mathrm{return}(x == 1?2 : 1) \end{aligned} \tag{19}$$

9.4 modify pointer

Some microcodes are subject to modifiers, which change the pointer that the microcodes act upon. The modifier function acts on the tuple $< p_{\mathbf{I}}, p_{\mathbf{F}}, p_{\mathbf{R}}, p_{\mathbf{W}} >$

```
$\mathrm{mp}\left(p^{a}_{\mathbf{X}}\right)$ :
    switch($s_{\mathbf{X}+1}$)
        case '**A**': return $p_{\mathbf{I}}$
        case '**B**': return $p_{\mathbf{R}}$
        case '**C**': return $p_{\mathbf{W}}$
        default: return $p_{\mathbf{X}}$
```
(20)

where $\mathbf{X} \in \langle \mathbf{I}, \mathbf{F}, \mathbf{R}, \mathbf{W} \rangle$.

9.5 template match position

Our microprograms make use of the alignment of template sequences to execute 'goto'-like control of program execution. Since templates are used to position pointers, we can use inexact template matches to introduce stochasticity into execution of the code of the molecule. Unlike the scenario of binding, the length of the best alignment is known in advance — it is the length of the template sequence that follows the instruction that is immediately upstream of it. For example, the sequence **$JAMES%^** has length $\lambda = 5$, the length of the sequence of template codes after the **$**. Rather than using the *maximum* length of the matched sequences, we always use the length of the template as the basis for our probability calculation, as in equation 13.

```
$\mathrm{apos}\left(p^{a}_{\mathbf{pF}}\right)$ :
    $\sigma, \lambda = \phi(s^{a_{\mathbf{F}}}, s_{\mathbf{F}} \ldots s_{\mathbf{F}+\mathrm{len}(\mathbf{F})}, \omega)$
    if $\sigma > \mathrm{rand}\,()$
        return $s^{a_{\mathbf{F}}}_{\beta}$
    else
        return $p_{\mathbf{F}}$
```
(21)

where $s^{a_{\mathbf{F}}}_{\beta}$ is the end of the alignment produced by ϕ. We can thus use the alignment score to derive probabilities of execution path selection for active instructions. A slight mismatch in the sequences that delimit an otherwise infinite loop would cause the loop to be exited with a certain probability each cycle. Such a loop could act as a stochastic timer for example, with the chance of escape from the loop being inversely proportional to the strength of the alignment.

9.6 template match score

The score of an alignment between an associated template and a string can also be used by microcodes. The function $\mathrm{asco}\,(p^{a})$ works in a similar manner to the way to $\mathrm{apos}\,(p^{a})$, by aligning an associated template with a string. The difference is that this function returns the probability of the alignment rather than the position:

```
$\mathrm{asco}\left(p^{a}_{\mathbf{pF}}\right)$ :
    return $P\left(\phi(s^{a_{\mathbf{F}}}, s_{\mathbf{F}} \ldots s_{\mathbf{F}+\mathrm{len}(\mathbf{F})}, \omega)\right)$
```
(22)

10 Microcodes

10.1 `A` to `Z`: Template codes

These are represented by characters from the uppercase alphabet, and are used to form the templates for the matching process detailed above. When a template code is executed by the instruction pointer, its only effect is to increment the instruction pointer:

$$p_{\mathbf{I}}^{a} := p_{\mathbf{I}}^{a} + 1 \tag{23}$$

10.2 `^`: `toggle`

Toggle changes which instance of the pair of pointers of a particular type is currently active. This has the effect of switching which of the two strings the pointer type is currently active upon:

$$\begin{aligned} a_{\mathrm{mp}(p_{\mathbf{F}}^{a})} &:= \mathrm{swap}\left(a_{\mathrm{mp}(p_{\mathbf{F}}^{a})}\right) \\ p_{\mathbf{I}}^{a} &:= p_{\mathbf{I}}^{a} + 1 \end{aligned} \tag{24}$$

In our replicase example, toggle is used to shift the active read pointer onto s_2 so that s_2 will be copied (see **`^B`** in region 4 of figure 5).

10.3 `$`: `search`

This operator moves $p_{\mathbf{F}}^{a}$ to a position corresponding to a template match $\mathrm{apos}\left(p_{\mathbf{F}}^{a}\right)$. Note that $p_{\mathbf{F}}^{a}$ can be positioned on the active or the passive string, depending on the value of $a_{\mathbf{F}}$. Only the string to which $p_{\mathbf{F}}^{a}$ points at the start of execution is searched. If a match exists, **`search`** then aligns the flow pointer with the end of that match. $p_{\mathbf{I}}^{a}$ is always incremented. Where identical scores are encountered, the first such score along the string is the winning match. If no template exists, $p_{\mathbf{F}}^{a}$ is positioned to the next instruction after the **`search`**. This function is not subjected to modifiers.

$$\begin{aligned} p_{\mathbf{F}}^{a} &:= \begin{cases} p_{\mathbf{I}}^{a} + 1 & \text{if there is no associated template for } p_{\mathbf{I}} \\ \mathrm{apos}\left(p_{\mathbf{F}}^{a}\right) & \text{if } r < \mathrm{P}\left(\mathrm{apos}\left(p_{\mathbf{F}}^{a}\right)\right) \\ p_{\mathbf{F}}^{a} & \text{otherwise} \end{cases} \\ p_{\mathbf{I}}^{a} &:= p_{\mathbf{I}}^{a} + 1 \end{aligned} \tag{25}$$

The search function is used in regions 3 and 6 of of our replicase molecule in figure 5 to move the flow pointer to the end of region 8.

10.4 `>`: `move`

This function shifts $p_{\mathbf{I}}^{a}$ to the position of $p_{\mathbf{F}}^{a}$, unless $p_{\mathbf{I}}^{a}$ is already in that position, in which case $p_{\mathbf{I}}^{a}$ is simply incremented. Note that if $p_{\mathbf{I}}^{a}$ is pointing to a different string to $p_{\mathbf{F}}^{a}$, then it must still be moved to the position of $p_{\mathbf{F}}^{a}$ (an implicit use of **`toggle`**). This instruction is commonly used to construct execution loops. **`move`** can move any of the pointers if the appropriate modifier is encountered:

$$
\begin{aligned}
\mathrm{mp}\left(p^a_{\mathbf{I}}\right) &:= \begin{cases} p^a_{\mathbf{F}} & \text{if } \mathrm{mp}\left(p^a_{\mathbf{I}}\right) \neq p^a_{\mathbf{I}} \\ p^a_{\mathbf{I}} + 1 & \text{otherwise} \end{cases} \qquad (26) \\
p^a_{\mathbf{I}} &:= p^a_{\mathbf{I}} + 1 \qquad \text{if } \mathrm{mp}\left(p^a_{\mathbf{I}}\right) \neq p^a_{\mathbf{I}} \text{ AND } p^a_{\mathbf{I}} = p^a_{\mathbf{F}}
\end{aligned}
$$

10.5 `?:` `if`

This function is a way of conditionally moving $p_{\mathbf{I}}$, depending on whether some test returns TRUE or FALSE. When TRUE or FALSE is returned, $p_{\mathbf{I}}$ is incremented twice or once respectively. It can be used to check whether some iterative task has been completed, or it can be used as a stochastic timer by using an inexact match score compared with a random number.

`if` has a number of different functionalities, depending on the presence of modifiers or an associated template. If there is no associated template, it can be used to check if $p_{\mathbf{R}}$ is pointing at a valid string symbol, or if it has incremented beyond the last character on a string.

If an associated template is present, **`if`** uses a template match to carry out the boolean test. Using equation 13, $p(\phi_{a,b})$ is calculated an compared with a random number r between 0 and 1. The score for the match is tested against a random number, and **`if`** returns:

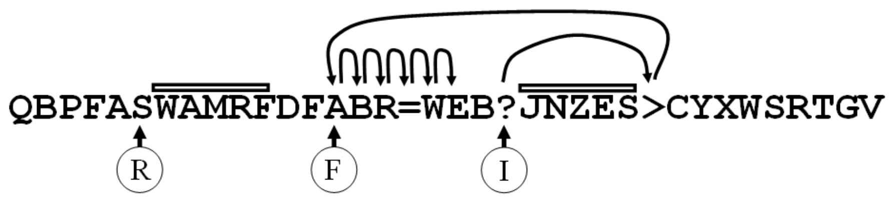
$$
p^a_{\mathbf{I}} := \begin{cases} p^a_{\mathbf{I}} + \mathrm{len}(p^a_{\mathbf{I}}) + 2 & \text{if } r <= \mathrm{P}\left(\mathrm{asco}\left(p^a_{\mathbf{R}}\right)\right) \\ p^a_{\mathbf{I}} + \mathrm{len}(p^a_{\mathbf{I}}) + 1 & \text{if } r > \mathrm{P}\left(\mathrm{asco}\left(p^a_{\mathbf{R}}\right)\right) \\ p^a_{\mathbf{I}} + 2 & \text{if there is no associated template for } p^a_{\mathbf{I}} \text{ AND } p^a_{\mathbf{R}} > \mathrm{len}(s^{a_{\mathbf{R}}}) \\ p^a_{\mathbf{I}} + 1 & \text{otherwise} \end{cases} \qquad (27)
$$

The function **`if`** can be used to escape from loops if a **`move`** command is one instruction downstream of the microcode. Figure 8 illustrates this. The **`if`** has the associated template **`JNZES`**. If $p_{\mathbf{R}}$ is not pointing at **`WAMRF`** (the complement of **`JNZES`**), then $p_{\mathbf{I}}$ is incremented to the **`>`** command, and then to $p_{\mathbf{F}}$ on the subsequent execution step. If $p_{\mathbf{F}}$ is upstream of **`?`**, the program is in a loop. The loop is escaped from when $p_{\mathbf{R}}$ is moved (via a **`=`** operator) to point to **`WAMRF`**. Only then does execution escape the loop becuase $p_{\mathbf{I}}$ is incremented two instructions downstream of the associated template. Region 5 in figure 5 carries out this sort of functionality to iteratively copy the partner molecule.

10.6 `=:` `copy`

This function inserts a copy of the code pointed to by $p_{\mathbf{R}}$ at $p_{\mathbf{W}}$. **`copy`** is commonly used to append the products of some reaction to the end of the executing string. If $p_{\mathbf{R}}$ is not pointing at a valid code, it has been incremented past the end of a string. If this is the case, no copy is carried out and the instruction pointer is simply incremented.

This instruction is the means by which strings increase in length. There exists a risk that programs can get trapped in loops such that they create infinitely long strings. To counter this,



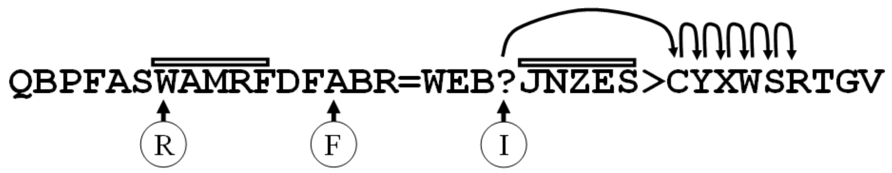


Figure 8: A program loop constructed using an **if** microcode. The instruction pointer follows a loop (shown top) while $p^a_{\mathbf{R}}$ does not match the associated template of $p^a_{\mathbf{I}}$. The loop terminates (shown bottom) when $p^a_{\mathbf{R}}$ points at a complementary match to the associated template of $p^a_{\mathbf{I}}$.

we specify a maximum string length Ω ($\Omega = 512$ in [8]). If either $p_{\mathbf{R}}$ or $p_{\mathbf{W}}$ go beyond this limit, **copy** acts as an **end** instruction.

There is scope to use modifiers to change which pointer **copy** reads from. This is not currently implemented. There is also scope to make the **copy** instruction be the container for mutation-on-**copy**. This will be the basis of our future Speigelman experiment [5].

```
if $p^a_{\mathbf{R}} < \Omega$ AND $p^a_{\mathbf{W}} < \Omega$
    if $p^a_{\mathbf{R}} < \mathrm{len}(s^{a_{\mathbf{R}}})$
        $s^a_{\mathbf{W}} := s^a_{\mathbf{R}}$
    $p^a_{\mathbf{I}} := p^a_{\mathbf{I}} + 1$
else
    end
```
(28)

10.7 **%: cleave**

This is used to split the string and generate a new molecule $\mathcal{M}^C$. The string is broken at $p_{\mathbf{F}}$, and the downstream portion is used to create a new molecule. If $p_{\mathbf{F}}$ is at the beginning or end of a molecule, no action occurs. Any pointers that were pointing at the region downstream of the cleave are moved to $p_{\mathbf{F}}$.

$$\begin{array}{lll} \mathcal{M}^C : s^1 & := & s_{\mathbf{F}} \\ s^{\mathbf{F}} & := & s^{\mathbf{F}}_0 \dots s^{\mathbf{F}}_{p_{\mathbf{F}}} \\ c & := & a_{\mathbf{F}} \\ & & \\ p^c_{\mathbf{I}} & := & p^a_{\mathbf{F}} \quad \text{if } p^c_{\mathbf{I}} > p^a_{\mathbf{F}} \\ p^c_{\mathbf{R}} & := & p^a_{\mathbf{F}} \quad \text{if } p^c_{\mathbf{R}} > p^a_{\mathbf{F}} \\ p^c_{\mathbf{W}} & := & p^a_{\mathbf{F}} \quad \text{if } p^c_{\mathbf{W}} > p^a_{\mathbf{F}} \end{array} \tag{29}$$

There is scope to use modifiers to change which pointer index **cleave** splits the molecule at. This is not currently implemented.

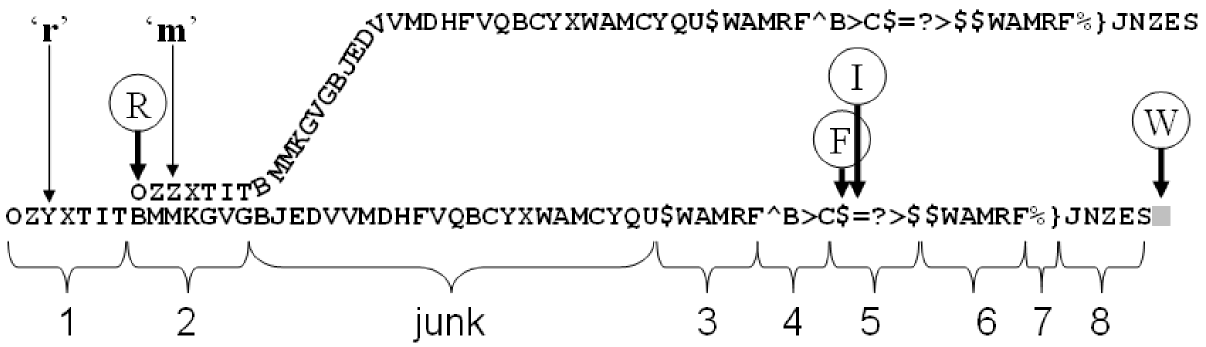

'r'
'm'
R
I
F
W
VMDHFVQBCYXWAMCYQU$WAMRF^B>C$=?>$$WAMRF%}JNZES
OZZXTITBMMKGVGBJEDV
OZYXTITBMMKGVGBJEDVVMDHFVQBCYXWAMCYQU$WAMRF^B>C$=?>$$WAMRF%}JNZES
1
2
junk
3
4
5
6
7
8

Figure 9: Composition of a replicase enzyme microprogram. The substrate sequence **m**(top) binds to the executing sequence **r** (bottom). There are eight distinct active regions of the molecule, plus one region of "junk" symbols. The sequence specifies: the initial molecular binding (regions 1 and 2); initialisation of the Read (R), Write (W) and Flow (F) pointers (regions 3,4 and 8); iterative copying of the substrate molecule (region 5); Repositioning the flow pointer to the end of the executing molecule (regions 6 and 8); cleaving off the newly created molecule and termination of the microprogram (region 7). Pointers, indicated as letters in circles, show the state of the microprogram as the first symbol of the template molecule is about to be copied to the end of the executing molecule. The two strings differ by a single mutation (indicated by thin arrows), allowing **m** to bind as a substrate to **r** more strongly than **r** binds to a copy of itself.

10.8 $\}$: `end`

This command terminates execution of the microprogram, and dissolves the bind between the active and passive molecules. The command is also called if $p_{\mathbf{I}} > \mathrm{len}(s)$, which occurs if the instruction pointer is incremented off the end of the string. There are no modifiers to this command, which reverses the binding process described in section 7.4.1. Firstly, the pointers and activators on the active molecule are reset:

$$\begin{aligned} p^1 &:= \perp \\ p^2 &:= \perp \\ s^2 &:= \perp \\ a &:= \perp \end{aligned} \tag{30}$$

secondly, the status of the two molecules in the pair is set to UNBOUND:

$$\begin{aligned} \mathcal{M}^1 : \delta &:= \mathrm{UNB} \\ \mathcal{M}^2 : \delta &:= \mathrm{UNB} \end{aligned} \tag{31}$$

Note that $\mathcal{M}^2 : s^1$ is not destroyed.

11 Experiments

Here we illustrate the system described above with an experimental evaluation of a replicase metabolism. This experiment was first described in [8]. Inspired by the RNA world model [17],

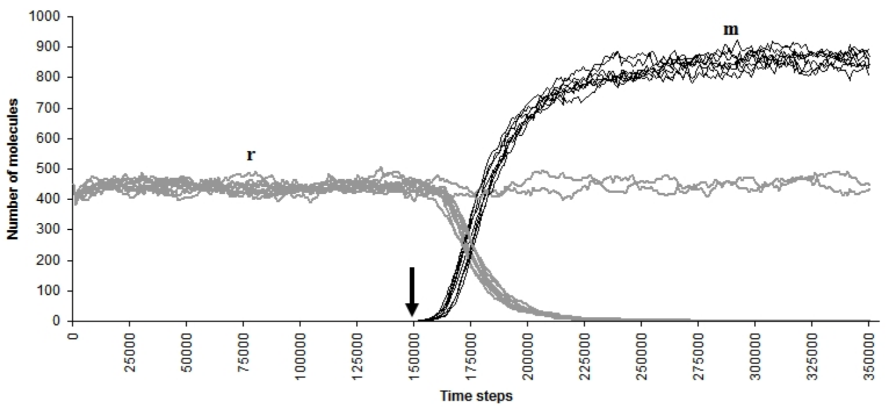


Figure 10: Invasion when rare. Ten typical runs of a metabolism of weakly binding replicase species $\mathbf{r}$ (thick grey lines) which is invaded (indicated by the arrow) by a single molecule of mutant species $\mathbf{m}$ (thin black lines), which has a stronger binding affinity. Eight of these trial runs demonstrate "invasion when rare" by $\mathbf{m}$. In two cases, $\mathbf{m}$ went extinct, and $\mathbf{r}$ remained at equilibrium.

we have conducted studies of metabolic systems composed of molecular microprograms without reference to a genome. In this framework, our molecular species act as their own templates. A replicating molecule is a good candidate for early trials, since only a single species needs to be defined. Our hand-crafted replicase $\mathbf{r}$ is shown in figure 9. Regions 1 and 2 of the microprogram for $\mathbf{r}$ are complementary sequences of match length $l = 7$, match score $s = 5.875$ and $\mathrm{P}\left(\phi\left(\mathbf{r}:s^1, \mathbf{r}:s^1, \omega\right)\right) = 0.293$ We have also hand-crafted a variant of $\mathbf{r}$ that could arise by a single mutation. This variant, $\mathbf{m}$, has perfectly matching complementary sites, so $l = 7$, $s = 7$, $\mathrm{P}\left(\phi\left(\mathbf{m}:s^1, \mathbf{m}:s^1, \omega\right)\right) = 1$, and importantly $\mathrm{P}\left(\phi\left(\mathbf{r}:s^1, \mathbf{m}:s^1, \omega\right)\right) = 1$ also: $\mathbf{r}$ binds to $\mathbf{m}$ more readily than $\mathbf{r}$ binds to $\mathbf{r}$. Note that there are very many species with replicase functionality in our molecular space.

Execution of the microprogram commences at the start of the bind and proceeds stepwise through each symbol to the right. The diagram shows the positions of the executing molecule's pointers (in circles) as the first symbol is about to be copied: $\mathbf{I}$ indicates that the next instruction is **=** (copy). $\mathbf{F}$ is set to the beginning of region 5, which executes the iterative copy process. $\mathbf{R}$ is positioned at the start of the template molecule's sequence. $\mathbf{W}$ is positioned at the end of the executing molecule's sequence. This is where the new molecule is built.

For experimental trials, we allowed a population of molecule $\mathbf{r}$ to reach equilibrium, then introduced a single molecule of $\mathbf{m}$ at time step 150 000. We ran a metabolism using this specification 100 times. A typical subset of 10 runs is shown in figure 10. The phenomenon of "invasion when rare" is occurring: the competitive advantage of species $\mathbf{m}$ allowed it to replace $\mathbf{r}$ entirely in 88 out of the 100 trials.

12 Conclusion

This document describes the first version of an artificial chemistry which allows binding and reaction function to be described by a sub-molecular sequence. Most of our efforts have been spent on defining a "baseline chemistry" and delimit the ingredients of the cell model and the generic reaction events. This has allowed us to build a modular cell model in which a chemistry appropriate for evolution can be developed.

We have shown that a replicator molecule can be built within this chemical model by building a replicase metabolsim. Although this is a useful first step, we now need to enlarge the model to accommodate more ambitious simulations.

References

[1] Hickinbotham, S., Clark, E., Stepney, S., Clarke, T., Young, P.: Gene regulation in a particle metabolome. In: CEC 2009, Trondheim, Norway, May 2009, IEEE Press (2009) 3024–3031

[2] Fischer, V., Hickinbotham, S.: A metabolic subsumption architecture for cooperative control of the e-puck. In: NICSO. (2010) accepted

[3] Clark, E., Hickinbotham, S., Stepney, S., Clarke, T., Young, P.: Encoding evolvable molecules. In: International Workshop on Information Processing in Cells and Tissues (IPCAT), Franscini Ascona, Switzerland, Librix (2009)

[4] Bentley, P.J.: Fractal proteins. Genetic Programming and Evolvable Machines (2004) 71–101

[5] Spiegelman, S., Kacian, D.L., Mills, D.R., Kramer, F.R.: A replicating rna molecule suitable for a detailed analysis of extracellular evolution and replication. Proceedings of the National Academy of Sciences **69** (1972) 3038–3042

[6] Johnson, T.J., Wilke, C.O.: Evolution of resource competition between mutually dependent digital organisms. Artif. Life **10** (2004) 145–156

[7] Ray, T., Xu, C.: Measures of evolvability in tierra. Artificial Life and Robotics **5** (2001) 211–214

[8] Hickinbotham, S., Clark, E., Nellis, A., Pay, M., Stepney, S., Clarke, T., Young, P.: Molecular microprograms. In: ECAL 2009 (LNCS 5777), Springer (in press)

[9] Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. J Mol Biol **147** (1981) 195–197

[10] Dill, K.A., Ozkan, S.B., Shell, M.S., Weikl, T.R.: The protein folding problem. Annual Review of Biophysics **37** (2008) 289–316

[11] Pennock, R.T.: Models, simulations, instantiations, and evidence: the case of digital evolution. J. Exp. Theor. Artif. Intell. **19** (2007) 29–42

[12] Ierymenko, A.: Nanopond: A very tiny artificial life virtual machine. `http://adam.ierymenko.name/nanopond.shtml` (2010)

[13] Bobrik, M., Kvasnicka, V., Pospichal, J.: Artificial chemistry and molecular Darwinian evolution of DNA/RNA-like systems I – typogenetics and chemostat. In Kelemen, A., Abraham, A., Liang, Y., eds.: Computational Intelligence in Medical Informatics. Volume 85 of Studies in Computational Intelligence. Springer (2008) 295–336

[14] You, L.: Toward computational systems biology. Cell Biochem Biophys **40** (2004) 167–184

[15] Shpaer, E.G., Robinson, M., Yee, D., Candlin, J.D., Mines, R., Hunkapiller, T.: Sensitivity and selectivity in protein similarity searches: A comparison of smith-waterman in hardware to blast and fasta. Genomics **38** (1996) 179 – 191

[16] Wikipedia: Smith-waterman algorithm, Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Smith-Waterman_algorithm` (2009) [Online; accessed 14-April-2009].

[17] Lincoln, T.A., Joyce, G.F.: Self-sustained replication of an RNA enzyme. Science (2009) 1167856+

A Worked example: processing string alignments

Section 6.4 described how our molecular analogues bind via a process of complementary Smith-Waterman alignment. To illustrate the process, let us consider the alignment of two strings:

s' :	**`NNNOUV>$PQVONNN`**
$C(s')$:	**`AAABHG>$CDGBAAA`**
s :	**`ABABABHG$CDBAAC`**

table 6 shows the matrices H and T for this alignment. The maximum value is 7.72, which is the *score* σ of the alignment. We have underlined this value, and underlined the trace back through the matrix by which the alignment is found. The trace through T is shown with large arrows.

									s							
			A	A	A	B	H	G	>	$	C	D	G	B	A	A	A
s'	C(s')																
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
N	A	0	1.00	1.00	1.00	0	0	0	0	0	0	0	0	0	1.00	1.00	1.00
O	B	0	0	0.88	0.88	2.00	0.67	0	0	0	0	0	0	1.00	0	0.88	0.88
N	A	0	$\underline{1.00}$	1.00	1.88	0.76	0.88	0	0	0	0	0	0	0	2.00	1.00	1.88
O	B	0	0	$\underline{0.88}$	0.88	2.88	1.55	0.22	0	0	0	0	0	1.00	0.67	1.88	0.88
N	A	0	1.00	1.00	$\underline{1.88}$	1.55	1.76	0.55	0	0	0	0	0	0	2.00	1.67	2.88
O	B	0	0	0.88	0.88	$\underline{2.88}$	1.55	0.88	0	0	0	0	0	1.00	0.67	1.88	1.55
U	H	0	0	0	0	1.55	$\underline{3.88}$	2.55	1.22	0	0	0	0	0	0	0.55	0.76
T	G	0	0	0	0	0.22	2.55	$\underline{4.88}$	$\underline{3.55}$	2.22	0.89	0	1.00	0	0	0	0
$	$	0	0	0	0	0	1.22	3.55	4.38	$\underline{4.05}$	2.72	1.39	0.03	0.75	0	0	0
P	C	0	0	0	0	0	0	2.22	3.05	4.26	$\underline{5.05}$	3.72	2.39	1.06	0.50	0	0
Q	D	0	0	0	0	0	0	0.89	1.72	2.93	4.14	$\underline{6.05}$	$\underline{4.72}$	3.39	2.06	0.73	0
O	B	0	0	0	0	0	0	0	0.39	1.60	2.81	4.72	5.17	$\underline{5.72}$	4.39	3.06	1.73
N	A	0	1.00	1.00	1.00	0	0	0	0	0.27	1.48	3.39	3.84	5.05	$\underline{6.72}$	5.39	4.06
N	A	0	1.00	2.00	2.00	0.88	0	0	0	0	0.15	2.06	2.51	3.72	6.05	$\underline{7.72}$	6.39
P	C	0	0	0.75	1.75	1.88	0	0	0	0	1.00	0.73	1.31	2.39	4.72	6.39	7.47
N	A		·	·	·										·	·	·
O	B			↖	↖	↖	←							·		↖	↖
N	A		·	·	↖	↖	↖								↖	·	↖
O	B			**↖**	↖	↖	←	↖						·	↑	↖	↖
N	A		·	·	**↖**	↑	↖	↖							↖	↖	↖
O	B			↖	↖	**↖**	↖	↖						·	↑	↖	↖
U	H					↑	**↖**	↖	←							↖	↖
T	G					↑	↖	**↖**	**←**	↖	←		·				
$	$						↖	↖	↖	**↖**	←	←	←	↖			
P	C							↖	↑	↖	**↖**	↖	↖	←	←		
Q	D							↑	↑	↖	↖	**↖**	**←**	↖	↖	←	
O	B								↑	↖	↑	↖	↖	**↖**	↖	↖	↖
N	A		·	·	·					↑	↖	↑	↑	↖	**↖**	↖	↖
N	A		·	↖	↖	↖					↑	↑	↑	↖	↖	**↖**	↖
P	C			↖	↖	↖					·	↑	↖	↖	↖	↖	↖

Table 6: A subsequence alignment H (shown top) generated from the scoring matrix in table 3. The alignment trace matrix T (bottom) contains all possible alignments. The final resulting alignment is shown with large arrows.

The trace through H and T produces the following alignment:

s' :	**`--NNNOUV>$PQVONNn`**
$C(s')$:	**`--AAABHG>$CDGBAAa`**
s :	**`abABABHG-$CD-BAAc`**

where characters outside the alignment are shown in lower case, and indels are indicated by a dash. Note also that the length of the alignment of each string is different, due to the indels.

B Version control summary

This is the baseline version of the Stringmol system, used in [8].