Embodied copying for richer evolution

Adam Nellis[1] and Susan Stepney[1]
[1]York Centre for Complex Systems Analysis (YCCSA), University of York, UK, Y010 5GE
adam@cs.york.ac.uk

Abstract

We address the process of *copying* in Artificial Life organisms. Copying is a source of mutations, a crucial component in evolution. We propose that rich copying mechanisms, and thereby rich evolutionary systems, can be obtained by *embodying* the copying process in a lower-level simulation.

We demonstrate an embodied copying process that has the potential to alter its own mutation rate, without having the concept of a mutation rate parameter explicit in the system.

Introduction

In computing, the concept of copying is important. Many programs copy data during computation. So programming languages often have the concept of copying as a primitive instruction. For example, all high-level imperative languages have an assignment operator; $a := b$ copies the contents of b and puts the result in a.

In Artificial Life (ALife), the concept of copying is also important. To reproduce, life-forms (whether biological or artificial) need to copy themselves. ALife organisms in computers can make use of the copy operations in programming languages, using these to copy themselves. But the requirements of ALife organisms and traditional computer programs are different. Copying in ALife is a source of mutations. It is a novelty-generation process driving evolution.

In biological organisms, copying is not an abstract concept implemented by a defined instruction. It is an emergent property of lower level processes. Copying is *embodied* within the biological systems that are being copied, and so mutations caused by the copying process can change the copying process.

We propose that ALife organisms should not blindly use the copy operations provided by programming languages. Here, we focus on copying as an embodied *process*, rather than as a computational *result*.

Artificial Chemistry (AChem) is the medium we use to embody the copying process. We explain how existing work has started to implement embodied copying reactions in AChems. We build on this by designing an AChem and using it to implement an embodied copying process.

Crisp, stochastic, and embodied copying

In normal computer programs, copying should happen *crisply*, without any errors. If a programmer writes $a := b$ in their code, they expect the copy to work perfectly. They expect a to contain an exact copy of b.

However, this is not the case in ALife. When biological life-forms (such as bacteria) clone themselves asexually, the clones are not exact copies of their parents (see any biology textbook, e.g. [1]). The biological 'copy operation' does not work perfectly. But this is not a mistake. Biology would not be improved by a perfect copy operation. Imperfect copying in biology causes the mutations and novelty that allow evolution to happen.

Stochasticity is a way of introducing variation into computer programs (or more generally, any systems). ALife organisms can use this variation to explore the design space of possible organisms. Stochastic programs are crisp programs with variation introduced via pseudo-random number generators. Stochastic programs can influence ALife organisms, allowing the organisms to vary. But the variation originates outside the simulation of the organisms, so the organisms can not influence the variation process. They can not change the stochastic programs. If the programs are to be changed during a simulation, they much be changed by another abstract process, operating on a higher level. This process, in turn, can only be changed by a process operating on an even higher level. This chain of meta-processes and meta-parameters can be broken by *embodying* the process in the simulation.

Embodying means implementing one system (the process) within another (the environment). It is frequently used in robotics to refer to building physical robots rather than simulated ones, thus embodying the robot system (process) in the physical world (environment). But the environment within which a process can be embodied is not limited to the physical world [6]. All processes are embodied within some environment, but stochastic programs are embodied in a trivial environment outside the simulation of the ALife organisms. This is why the ALife organisms can not change the stochastic programs.

Algorithm 1 Deconstructing the string copy operation, as a prerequisite to embodying it

```
result string_A := string_B

i := start(string_B)
while i not at-end(string_B) do
  string_A(i) := char-copy(string_B(i))
  i := next(i)
end while
```

In order for an ALife organism to change a stochastic process, the process must be implemented in the same language as the organisms: the process must be embodied within the simulation of the organism. A stochastic copying process allows the organisms in a simulation to vary, and so evolve. If the copying process is itself part of the simulation, then it too will be able to vary and evolve.

Copying as a process

In writing an embodied copying program, we must think about the *process* of copying a string, rather than the *result* of the copy. Algorithm 1 breaks down this process into four parts, each involving a particular function: `start`, `at-end`, `char-copy`, and `next`. Each of these four functions can be either crisp, stochastic or embodied. If all four are crisp, then the overall copying process is crisp, and exact copies are always produced.

If any of these four functions are stochastic, then the overall copying process will be stochastic. Making different combinations of these four functions stochastic introduces different kinds of variation into the copying process. For example: making `char-copy` stochastic could cause some characters to be copied incorrectly; making `at-end` stochastic could cause the copy to be truncated.

We can embody the copying process in different ways, and to different degrees. We must implement a simulation of a system where at least one of these functions can happen as a consequence of lower-level events. But we do not need to embody all four of the functions. We can implement some of them as crisp or stochastic functions in the definition of our simulation. Thus there are many different ways in which we can embody the copy operation. Each of these ways leads to different systems with different properties and different degrees of self-modification and novelty generation.

Example: the Stringmol AChem

The Stringmol AChem [3, 2, 4] has been used to implement an embodied copy operation. In terms of algorithm 1, it has embodied `start` and `at-end` functions, a stochastic `char-copy` function and a crisp `next` function.

Stringmol's embodied copying process has been shown to produce interesting behaviour [3]. Because the process of copying is embodied in a 'replicase' chemical, evolution can change the process when the replicase copies (another instance of) itself. One sequence of changes observed in Stringmol (described in detail, in [3]) is the emergence of an unprogrammed 'macro-mutation' that chops off the first few characters of a chemical. The emergence of the macro-mutation exploited the two embodied stages of Stringmol's copying process: the `start` and `at-end` functions.

Stringmol produced something different from what would normally be expected of a copy operation: an unprogrammed type of mutation. The emergence of a new type of mutation is not possible using just a stochastic copy operation. Embodiment is needed to allow the intermediate stages of the copying process to be exploited and changed. This shows the potential power of embodying the copying process (or more generally, any process).

Our hypothesis is that by embodying different stages of the copying process, we will be able to observe different, unprogrammed types of mutation emerging from our ALife simulations.

The Graphmol AChem

In our Graphmol AChem, the chemicals are graphs, and reactions change the topology of the graphs. We use Graphmol to build an embodied copy operation that has an embodied `next` function. Here we use crisp `start`, `at-end` and `char-copy` functions, because we are interested in investigating the effect of embodying the `next` function. However, Graphmol has been designed so that the `start`, `at-end` and `char-copy` functions can (in the future) be made stochastic or embodied.

We embody the `next` function by building a "walker" chemical in Graphmol. This chemical is a graph that can change its own topology by running short computer programs. Some of its graph nodes are "feet" that walk along the string being copied (which is also represented as a graph). The `next` function (incrementing a pointer) is broken down into two stages: (1) lifting up a foot; and (2) putting that foot down in the 'next' place. A stochastic process controls where the feet are put down, allowing them to be put down in the "wrong" place and so causing mutations in the copied string (variations in the copying process). The `next` function is embodied because the stochastic process depends on the composition of the walker chemical. Changing the walker chemical changes the stochastic process, and so an evolving walker chemical can change the way in which it performs its `next` function.

We show that this embodiment allows the walker chemical to change its mutation rate through evolution. This demonstrates the usefulness of an embodied `next` function (increment operation) used to make an embodied copy operation.

[	begin	Begin defining a binding site
]	end	End definition of a binding site
<	show	Show binding site
>	hide	Hide binding site
!	stop	Stop the execution of a program
a-z, 0-9	junk	Non-functional atoms

Figure 1: The alphabet of Graphmol atoms.

Definition of Graphmol

The chemicals in Graphmol are represented by graphs. Each graph is both a data structure and a program. The execution of the program changes the structure of the graph.

A Graphmol chemical is defined by a string of atoms over an alphabet (figure 1). This string is parsed into three types of nodes (binding sites, which can be `shown` or `hidden`; functions; and junk), and folded into a graph with three types of edge (program edges, fold edges, and bind edges). Distances through the graph are used in a stochastic binding process (distances are calculated using the number of atoms in each node).

Reactions are defined by the Graphmol programming language, which has two parts: a declarative part (binding process) and an imperative part (instruction pointers).

The declarative part defines how chemical graphs bind to each other, implemented by a simple aspatial physics engine. This continually changes the graph structure by adding bind edges between `shown` binding sites. The process is stochastic, and the chance of two binding sites binding (having a bind edge added) depends on: (1) how closely their binding site patterns match; and (2) their distance apart, through the graph (measured as the length of the shortest path between the two binding sites).

The function nodes in the graph are the imperative language instructions. Instruction pointers move through the graph, executing the function nodes. This changes the graph structure by `showing` and `hiding` binding sites. When binding sites are `shown`, new binds become possible; when binding sites are `hidden`, some binds become impossible.

Junk affects how programs run in two ways: (1) it acts as a no-op for instruction pointers moving through the graph, slowing down execution of programs with respect to the timescale of the binding process; (2) it affects the graph distance between nodes, used to calculate binding probabilities.

Parsing and Folding There are two steps in converting a string of atoms into a chemical graph. These are (1) parsing atoms into nodes and (2) folding: connecting function nodes to their binding site nodes (figure 2).

A sequence of non-functional atoms enclosed in brackets `[nnnnn]` (with no internal brackets) defines a *binding site*. So the string `[hdfggd[icsd]bdgd[dhdhd]ixr]ss` defines two binding sites, `icsd` and `dhdhd`. The string of

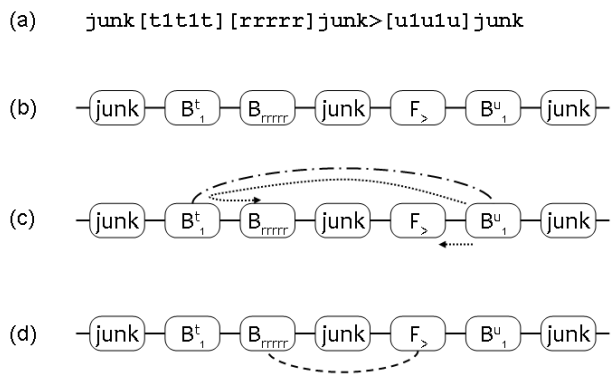


Figure 2: Parsing and folding: (a) a string of atoms; (b) the parsed graph of nodes connected by program arcs (solid edges); (c) temporary edge between `u1u1u` binding node and `t1t1t` binding node (dash-dotted edge), used to find closest functional node and binding site (dotted arrows); (d) resulting fold edge between the function node and its binding site (dashed edge).

atoms is parsed into a linear graph (figure 2 (b)) of binding site nodes, function nodes, and junk (everything else).

When executed, each < or > function `shows` or `hides` a particular binding site. The folding process connects these functions to the sites they affect. A temporary graph edge is added between a binding site node of the form `uxuxu` and its matching `txtxt` binding site node (where `x` is any non-functional atom). The closest `show` or `hide` function node, and closest the binding site node (measured along edges including the temporary edge), to the `uxuxu` node, are joined by a fold edge, and the temporary edge is removed. The result is the folded chemical graph (figure 2 (d)).

The fold edge is a form of indirect addressing. Instead of the function note specifying explicitly which binding site it `shows` (or `hides`), it instead specifies a template: `uxuxu`. During folding, this template is 'dereferenced' to locate the binding site: the closest binding site to the matching `txtxt`. Indirect addressing makes the system more evolvable, because the templates can change independently of the pattern of the target binding site.

Chemicals Once strings have been parsed and folded into chemical graphs, the graphs can start to react. The physics engine starts binding matching sites (here, we use exact string matching, so two binding sites either match or they do not; binding is a crisp process). When more than two sites match, the choice of which to bind is made stochastically, based on the graph distances between the sites.

When two binding sites bind, a bind edge is created between them, and an instruction pointer is created at each binding site. These instruction pointers move along their respective chemicals, executing any functions they reach.

As Graphmol runs, the graph states change because of two

processes: (1) instruction pointers move along the chemicals, executing functions that `show` and `hide` binding sites; (2) the physics engine makes binds happen between binding sites that match and are close together, which creates new instruction pointers.

Reactions There are two different concepts of 'reaction' in Graphmol: (1) a micro-scale interaction between two binding sites; (2) a macro-scale interaction between two chemicals, either designed into the chemicals, or an emergent property of the system.

(1) In the micro-scale case, a 'reaction' is the same as a bind between two binding sites. If two binding sites have patterns that match, then they have a probability of binding that depends on their distance apart through the graph. If the two binding sites are on different chemicals (that are not bound), then their distance apart is not defined, and they have a (pre-specified) low probability of binding.

When a bind happens, a bind edge is created between the two binding site nodes. This changes the topology of the chemical graphs, changing the probabilities of other binds happening. This new edge remains in place until one of its binding site nodes is `hidden`, at which point the edge is removed. When the bind happens, two instruction pointers are created, one at each binding site node. They move along their respective chemical's program edges, executing any function nodes they encounter, until they reach either the end of the chemical, or a stop, (`!`), function, at which point the instruction pointer is removed.

The immediate result of this type of reaction is a graph topology change. The two chemicals are now connected together, and so the distances between binding sites have changed. A longer-term result of this reaction is that two computer programs are now running, represented by the two instruction pointers that are created. If another bind happens before these programs finish running, then further programs start executing in parallel.

This definition of 'reaction' views Graphmol as a simulation of nodes in a graph. Graphmol simulates these nodes by continually iterating the instruction pointers that exist (running the programs), and checking if any new binds happen (starting new programs). As the programs run, new binding sites become visible and so new binds can happen.

(2) In the macro-scale case, a 'reaction' is not defined explicitly as part of the Graphmol program: instead, it is a property of a running system. This can be an emergent property, produced by an evolutionary system. But in order to bootstrap evolutionary systems, we can design macro-scale reactions by hand-crafting Graphmol chemicals.

In traditional AChems, a reaction is a process whereby two chemicals are chosen to enter a black box, something happens, then one or more chemicals emerge from the box. Viewing Graphmol as a simulation of graph nodes does not fit this black box definition of a reaction. But we can use the simulation to implement *white* box reactions instead.

```
[start] junk
[llllll] ! junk [xxxxx] ! junk [rrrrr] ! junk
[llllll] ! junk [xxxxx] ! junk [rrrrr] ! junk
...
[llllll] ! junk [xxxxx] ! junk [rrrrr] ! junk
[llllll] ! junk [xxxxx] ! junk [rrrrr] ! junk
[stop]
```

Figure 3: The Graphmol DNA as a string of atoms (whitespace added for readability only). The `xxxxx` binding sites are the bases that carry the information. The DNA chemical can be of arbitrary length.

We can design two chemicals that have binding sites with matching patterns. We can set up the internal states of these chemicals so that only the two matching binding sites are `shown` (the rest being `hidden`). When we put these chemicals into the simulation, they will bind and start executing their programs. The execution of their programs might cause other binding sites to become `shown` and other binds to happen, but eventually all the programs will stop and no more binds will be possible. The individual programs cannot go into an infinite loop, since they execute along the program edges of a linear graph. The whole simulation could go into an infinite loop, but we assume not, for this argument.

We can think of this whole process as one 'reaction', and the system now looks like a traditional AChem, but with a complicated reaction mechanism. The chemicals that now exist in the simulation are the products of the 'reaction'. Macro-reactions of this type are white boxes, because they are embodied in the simulation. This means that other chemicals can interfere with the process of the reaction.

Embodied copying in Graphmol

Binding and program execution change the topology of chemical graphs. We use this to make one chemical graph move, relative to another. We make a long linear chemical graph composed of binding sites separated by regions of junk. This chemical contains no function atoms, so will not change its own topology. We make a second, smaller, chemical that 'walks' along the long chemical by alternately `showing` and `hiding` its six binding site 'feet'. We add a special crisp `char-copy` instruction to the Graphmol language, specifically for the purpose of the experiments reported here.

The idea of a small chemical moving along a long, linear chemical is analogous to the way in which DNA is copied in biology. DNA is a long linear chemical. The chemical 'DNA polymerase' moves along the DNA and copies it. The actual process in biology is much more complicated than this, but making a simplified abstraction of the process allows us to implement an embodied copy operation in an AChem. Furthermore, many chemicals in biology move along DNA or RNA chemicals (not just to copy them). For example: (see

```
[t1t1t] [yyyyy] junk >[u4u4u] junk <[u2u2u] !
[t2t2t] [magic] junk >[u5u5u] junk <[u3u3u] !
[t3t3t] [eeeee] junk >[u6u6u] junk <[u4u4u] !
[t4t4t] [yyyyy] junk >[u1u1u] junk <[u5u5u] !
[t5t5t] [magic] junk >[u2u2u] junk <[u6u6u] !
[t6t6t] [eeeee] junk >[u3u3u] junk <[u1u1u] !

[tstst] [fgneg] junk <[u1u1u] junk >[ususu] !

[tetet] [fgbc]  junk junk junk junk
     junk >[u1u1u] junk >[u2u2u] junk >[u3u3u]
     junk >[u4u4u] junk >[u5u5u] junk >[u6u6u]
     junk <[ususu] junk >[ueueu] !
```

Figure 4: The Graphmol walker as a string of atoms. There are six feet (`t1t1t`–`t6t6t`), a 'start' site (`tstst`) and a 'stop' site (`tetet`). The length of the junk sections is varied in the experiment (see later).

any biology textbook for details, for example [1]) helicases (that unwind the two strands of DNA), ligases (that glue together sections of DNA) and ribosomes (that transcribe RNA into protein).

So, if we are interested in simulating analogies of biology, then movement of one chemical along another is a useful type of process to have in general.

Graphmol DNA We design a Graphmol chemical analogous to biological DNA. DNA stores information as a sequence of DNA bases attached to a common "backbone" structure.

Graphmol DNA has a sequence of 'base' nodes containing different information, interspersed with backbone nodes (figure 3). A 'base' node is a binding site, whose patter is five information-carrying atoms (shown generically as `xxxxx`). Two backbone nodes `[llllll]` and `[rrrrr]` give the DNA a direction. (The stop atoms, `!`, are for efficiency, to remove the instruction pointer that is created on the DNA when a bind occurs.)

The junk regions add distance between the binding sites, which controls the probability of binding to different sites. In the implementation reported here, the DNA's junk regions are each 40 atoms long.

The DNA chemical also has a `start` and a `stop` binding site. These allow the walker chemical to begin copying from the start of the DNA and to unbind when it reaches the end. This allows us to program the copy operation as a 'macroscale reaction', as described above.

Graphmol Walker The walker chemical is shown in figure 4 as a sequence of atoms; its walking behaviour is shown schematically in figure 5. The walker chemical moves along the DNA chemical using six 'feet' (binding sites) alternating their visibility in a cycle. Feet 1 and 4 bind to `[llllll]` on the DNA, feet 2 and 5 to `[xxxxx]`, and feet 3 and 6 to `[rrrrr]`. In this paper, binds happen if sites match exactly, where alphabet atoms match their complements (rotated 13 characters through the alphabet), and digits do not match.

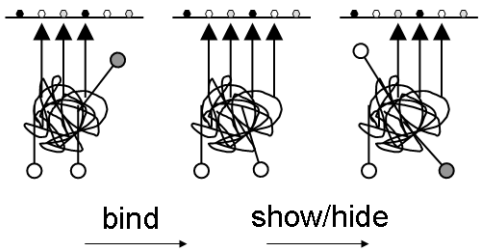


Figure 5: The walking process. Feet 1–3 are shown and bound (triangles), foot 4 is shown and unbound (dark circle), feet 5 and 6 are hidden (white circles). **bind:** The physics engine binds foot 4 (which is now shown with a triangle). **show/hide:** The bind starts a program running, which hides foot 1 (which therefore unbinds), and shows foot 5 (which is unbound). The cyclic process is ready to start anew.

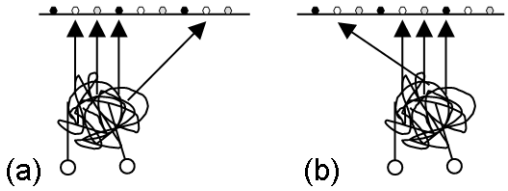


Figure 6: Low probability mis-stepping: (a) stepping over a site; (b) stepping backwards.

For the purposes of this paper, the Graphmol language is extended with `magic` binding sites that match and bind to any of the DNA's `[xxxxx]` information-carrying binding sites. It performs a crisp copy of the bound node (a crisp `char-copy` function, from algorithm 1).

The walker has a 'start' `[fgneg]` region and a 'stop' `[fgbc]` region. The start region sets up the walker's feet ready to begin moving along the DNA. The end region unbinds the walker from the DNA and sets the walker up ready to start another copy. This is a crisp `start` and `at-end` function, from algorithm 1.

Each foot has a short program associated with it. These programs `show` and `hide` the walker's six feet in a cyclic pattern, making it walk along the DNA (figure 5).

Each of the walker's feet has a pattern that matches multiple binding sites on the DNA. Because the probability of binding depends stochastically on graph distance, the walker's feet will always be more likely to bind to sites on the DNA that are close to where the walker is currently bound. As three of the walker's feet are always bound at the same time, the next matching binding site along the DNA will always be closer to the walker's shown foot than earlier or later DNA sites. The walker usually steps to the correct next binding site, but can sometimes (with a low probability controlled by the amount of junk) jump forwards or step backwards (figure 6). Thus the walker implements an embodied `next` function (from algorithm 1).

Because the walker is copying the chemical it walks over. These jumps forwards and backwards correspond to insertions and deletions in the copied chemical.

Through the same binding process, the walker can also occasionally get its feet tangled, and fall off, resulting in a truncated copy. So the walker also implements an embodied `at-end` function. It has two `at-end` functions: a crisp one ('stop' region) and an embodied one (fall off early).

Experiment

The Graphmol walker chemical, described in the previous section, can copy a DNA chemical, making insertion, deletion and truncation errors. But the way in which it makes these errors is not a collection of arbitrary choices written into an equation or a piece of stochastic code. It is a collection of arbitrary choices written into a machine (chemical) implemented in a lower-level stochastic programming language (Graphmol). If this machine/language combination is evolvable, then these arbitrary choices can be changed by evolution, and adapted to the problem being solved.

This paper is a feasibility study, testing that the embodied copying process implemented by the walker is evolvable. We show that, due to the design of the walker and of Graphmol, there is evolutionary pressure for the walker to evolve. It can trade off its accuracy against its speed of copying, by altering its level of junk. With more junk, the walker copies more accurately but also more slowly. With less junk, the walker copies less accurately but also more quickly.

Experiment design

We want to test the hypothesis that changing the walker's junk level changes its speed and accuracy of copying.

To test this hypothesis, we run multiple simulations of the walker copying the DNA chemical. The length of the DNA chemical (number of bases) is the same as the length of the walker (number of atoms). This simulates the fact that if the walker was evolving, then changing its junk level would change the length of its encoding on the DNA.

We set up the DNA chemical by `showing` all of its binding sites. We set up the walker chemical by `hiding` all of its binding sites except the 'start' site `[fgneg]`. We then bind the walker's 'start' site to the DNA's 'start' site and simulate the (macro-scale) reaction until the walker unbinds from the DNA, thus finishing its copy. When the walker unbinds from the DNA, we compare its copy to the original DNA. The pattern of bases on the original DNA is randomly generated each time.

We repeat this copying process for walker chemicals containing different levels of junk. The junk regions in the walker chemical (see figure 4) are varied in length from one atom to 20 atoms. In this experiment, all of the junk regions within the walker are the same length as each other, for simplicity. If the walker was evolving, it would not need to enforce this. Indeed, unless there was evolutionary pressure for it, evolution would probably not maintain 26 different regions at the same length. So this experiment shows a coarse view of the evolutionary options the walker has. In reality, the walker has a much finer level of control over its junk regions than this experiment shows.

For each different level of junk, we measure the time taken for the walker to make a copy (figure 7(a)) and the accuracy of its copying (figure 7(b)). Since the walker can make insertions, deletions and truncations of the DNA it is copying, there are many ways to define accuracy. We use the following. We care about the walker copying the DNA almost perfectly: we want perfect copies most of the time, but occasionally we want small mutations for evolution to exploit. So we define an 'almost perfect copy' as a copy that differs from the original by at most three bases, i.e. any combination of three insertions or deletions. To determine if a copy is almost perfect, we use Smith-Waterman alignment [5]. The Smith-Waterman algorithm measures the length of the longest common subsequence between two strings, taking into account (and penalising) short insertions and deletions. We set the penalty for an insertion or deletion to be 1, to measure the number of errors in the copy (subtracting the length of the original DNA, and taking the absolute value). If the number of errors is three or less, the copy is 'almost perfect'. Values other than three give qualitatively similar results, but larger values are more noisy so more experiments would need to be run to obtain the same error bars.

We run 80 copies per junk level, counting the number of nearly perfect copies to measure accuracy. We then repeat this process 20 times, to determine the error in these measurements (shown as notched boxplots in figure 7).

Results

As the junk level increases, the walker takes longer to copy its DNA (figure 7(a)). This is for two reasons: (1) more junk makes the graph distance between binding sites longer, so the probability of the walker binding (and hence taking a step) is reduced; (2) more junk means the walker's encoding on the DNA is longer, so takes more steps to copy.

As the junk level increases, the walker becomes more accurate at copying its DNA (figure 7(b)). This is in spite of there being more DNA to copy at higher junk levels. As junk increases, the probability of binding is reduced in such a way that the probability of an erroneous bind (either jumping forwards or stepping backwards, figure 6) is reduced more than the probability of it making a correct bind (the probability is a non-linear function of distance, $p(d) = (20/d)^{77}$, chosen to give good behaviours over a range of chemical sizes). This makes the walker more accurate with more junk.

A walker with a low junk level is fast but error-prone; a walker with a high junk level is slow but reliable. So, the walker can trade off accuracy against speed. We can see this tradeoff by graphing the rate of copying for each junk level (figure 7(c)). This is the number of nearly perfect copies

made, divided by the time taken to make them. The graph is noisy at low junk levels because few nearly perfect copies are made here (as can be seen from the accuracy graph, figure 7(b)). The tradeoff can be seen in this graph as a peak at a moderate amount of junk. Too much junk and the walker copies too slowly, making its *rate* of accurate copying low. Too little junk and the walker makes too many errors, making its rate of *accurate* copying low.

Discussion

When the walker is put into a simulation where it can evolve, it will be able to control its own junk level through mutations that add or remove junk. These results show that changing the walker's junk level changes its speed/accuracy tradeoff for copying. Thus the walker will be able to find, for itself, the tradeoff between speed and accuracy that optimises its survivability in its environment.

Because it finds this tradeoff for itself, it will be able to re-optimise if its environment changes. We have taken a quantity that is normally a parameter in ALife simulations, the mutation rate, and embodied the process that requires this parameter. This means that the ALife organisms can change this parameter, by manipulating the underlying processes that give rise to the parameter. The mutation rate has changed from being an external parameter, to an observed property of the system.

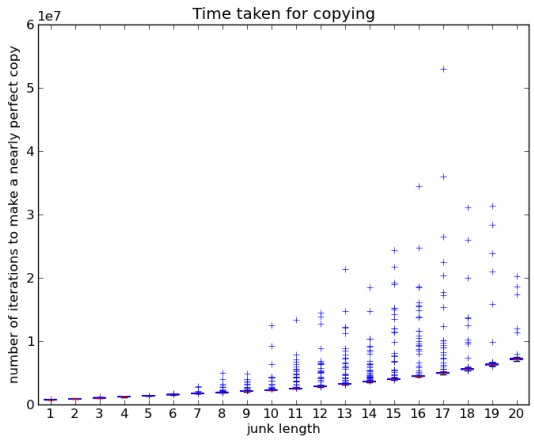
Future work

This experiment has demonstrated that it is possible to build an AChem with an embodied copying process that can be exploited by the system to adapt its mutation rate. But because the whole *process* of mutation is embodied (not just the rate), the system should be able to change the copying process, generating novel *types* of mutation. When we run the embodied copying process in a evolutionary system, we will be looking for such changes.

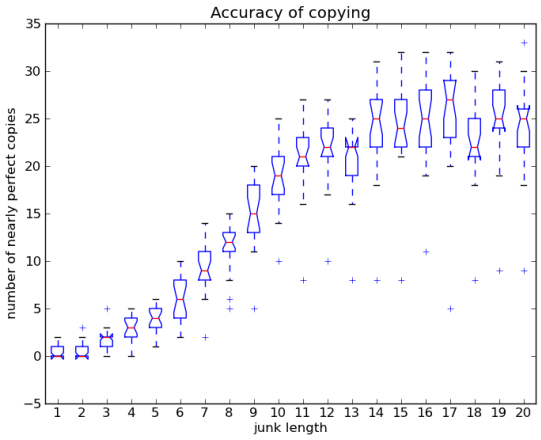
To make systems that can change their mutation process in different ways, different parts of the copying process can be embodied:

Copying a character

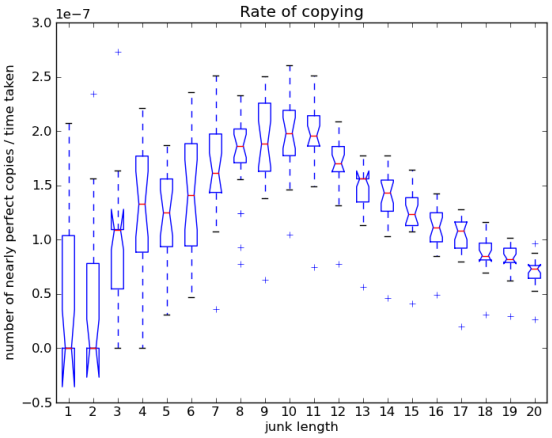
The walker chemical takes the process 'iterate over a string' and implements this as an embodied process in Graphmol. Here we have used a crisp `char-copy` function to copy each character of the string (so the only copying errors are insertions and deletions). The `char-copy` function could instead be made stochastic, to explore the effect of point mutations on the walker. More interestingly, the `char-copy` process could be embodied, by implementing a `char-copy` mechanism in Graphmol. Just as we broke the string copy process into components (algorithm 1), we can break the character copy process into components to be embodied (algorithm 2):



(a) Copying time increases with junk level (so speed decreases).



(b) Accuracy of copying increases with junk level.



(c) Rate of copying has an optimum junk level, trading off speed against accuracy.

Figure 7: How the walker's junk length affects its copying. A "nearly perfect copy" is a copy that differs from the original by at most three bases (three insertions or deletions). The notches show the 95% confidence intervals.

Algorithm 2 Deconstructing the character copy operation, as a prerequisite to embodying it

result char_A := char_B

x := `read`(char_B)
y := `repn`(x)
char_A := `write`(y)

1. **Read the character** `read` This could be implemented as a set of binding sites with patterns that match each of the DNA bases. When one of these sites binds to a DNA base, a program on the walker changes the walker's state.

2. **Represent the character** `repn` The walker needs to know which site has been bound, so needs a change of state to signify this. For example, it could show a binding site corresponding to the base that it is currently copying.

3. **Write the character** `write` The walker needs to maintain a chemical representing the copy it is making of the DNA chemical. The binding site it shows in the step above, could bind to a free-floating chemical base, at which point the walker would attach this base to the copy.

After attaching the copied character to the result string, the walker needs to move on to the next character on the string being copied. The walker already does this to walk along a DNA chemical, but it will also need to do this with the copy it is producing.

Making binding evolvable

In this paper, binding requires an exact (complementary) match between binding sites. We need to allow 'soft' binding, so that evolution can modulate binding affinities to give complex behaviours [3]. This will make the `start` and `at-end` functions embodied (and `char-copy`, if used with the previous section).

In this paper, the folding sites `[uxuxu]` and `[txtxt]` (figure 2a) were chosen arbitrarily. In future, the folding process will be implemented by a folding chemical, with binding sites matching `[uxuxu]` and `[txtxt]`. This makes the folding process embodied, rather than hardcoding folding in the definition of Graphmol. Thus evolution will be able to exploit the folding process and potentially change it.

Conclusions

We have discussed how ALife simulations can be made more evolvable by making their copying process embodied rather than stochastic. An existing example of where an embodied copying operation has led to interesting behaviour is the Stringmol AChem [3].

We have embodied copying in a new way, by making an embodied `next` function (increment operation). This involved designing the Graphmol AChem and implementing an embodied `next` function in Graphmol. We attached a crisp `char-copy` function to this embodied `next` function, creating an embodied string copy operation. This embodied string copy operation can make insertion and deletion mutations on the copied string.

We have run a feasibility experiment (figure 7) to show that the embodied copy operation is evolvable, and has the potential to adapt its own mutation rate to its environment. But more experiments are needed to find the environments in which it will show this. The copying process adapts by changing the level of junk in its embodiment, which changes its probability of incrementing correctly versus incrementing erroneously. In this way, the embodied copy operation can adapt its own mutation rate without there being an explicit mutation rate parameter in the system.

This is the crucial difference between embodied systems and stochastic systems. Stochastic systems are crisp systems with parametrised variation added in. Embodied systems are evolvable machines that can evolve their own parameters and processes, because they are implemented in a lower-level language.

Acknowledgements

This work is part of the Plazzmid project, EPSRC grant EP/F031033/1. We thank the other members of the Plazzmid project for valuable comments and ideas: Ed Clark, Simon Hickinbotham, Peter Young, Tim Clarke and Mungo Pay. And the anonymous referees for their helpful comments.

References

[1] T. Brown. *Genomes 3*. Garland Science, 2006.

[2] S. Hickinbotham, E. Clark, S. Stepney, T. Clarke, A. Nellis, M. Pay, and P. Young. Molecular microprograms. In *ECAL 2009, Budapest, Hungary, September 2009*. LNCS. Springer, Sept. 2009.

[3] S. Hickinbotham, E. Clark, S. Stepney, T. Clarke, A. Nellis, M. Pay, and P. Young. Diversity from a monoculture: Effects of mutation-on-copy in a string-based artificial chemistry. In *ALife XII*, pages 24–31. MIT Press, 2010.

[4] S. Hickinbotham, E. Clark, S. Stepney, T. Clarke, A. Nellis, M. Pay, and P. Young. Specification of the stringmol chemical programming language version 0.1. Technical Report YCS-2010-457, Dept Computer Science, University of York, 2010.

[5] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[6] S. Stepney. Embodiment. In D. Flower and J. Timmis, editors, *In Silico Immunology*, chapter 12, pages 265–288. Springer, 2007.