# Self-Replicating and Self-Modifying Programs in Fraglets

Lidia Yamamoto
Computer Science Department
University of Basel
Bernoullistrasse 16
CH-4056 Basel, Switzerland
lidia.yamamoto@unibas.ch

Daniel Schreckling
Computer Science Department
University of Hamburg
Vogt-Koelln-Str. 30
D-22527 Hamburg, Germany
schreckling@informatik.uni-hamburg.de

Thomas Meyer
Computer Science Department
University of Basel
Bernoullistrasse 16
CH-4056 Basel, Switzerland
th.meyer@unibas.ch

## ABSTRACT

The inherently decentralized nature of artificial chemical computing models makes them particularly attractive for building bio-inspired software with self-organizing and emergent properties. Yet it is not straightforward to construct such chemical programs, either manually or automatically.

We are exploring the potential of chemical programming models for automatic programming, in the context of autonomic environments where software must operate unsupervised for unlimited periods of time. We are enhancing the Fraglets chemical language to support intrinsic genetic programming, such that programs can replicate and modify themselves during execution.

The Fraglets language was originally designed to express communication protocols. We first show a few extensions towards more generic computations, then show how self-replicating and self-modifying programs can be created. This is a first step towards programs that can repair and optimize themselves in an autonomic way. We reveal a number of features and shortcomings of the language, suggesting fixes and future directions.

## Keywords

Artificial Chemical Computing, Self-Modifying Code, Self-Replication, Quines

## 1. INTRODUCTION

Artificial chemical computing models are gaining increasing prominence in the design of bio-inspired software with self-organizing and emergent properties [2, 4, 8, 11], The inherently parallel and decentralized nature of the chemical computing metaphor makes it an attractive alternative to classical programming methods.

We are exploring the potential of chemical programming models in the context of autonomic environments where software must operate unsupervised for unlimited periods of time. Systems in these environments must be able to autonomously interact with the external world and among themselves. They must also be able to detect and repair their own failures, and to adapt their behaviour to new situations. Ultimately, the system should be able to reprogram itself, creating new behaviours and functionalities. The long-term goal is to obtain an intrinsic, code-level self-healing and self-optimizing system. The self-healing ability refers to detecting and repairing failures in the system's own code base, during execution. The self-optimizing ability implies the online evolution of its own code base, which could be envisaged via a resilient form of genetic programming.

The first step towards such long-term goal is to have a programming language that facilitates code self-modification. We are currently working with the Fraglets language [32], in which code and data are represented as virtual molecules that are transformed using a chemical reaction metaphor. These virtual molecules represent computation fragments that can be easily dispersed over several nodes in a network, offering a natural model for self-replicating and mobile code. Fragments are consumed and produced during execution, thus the code is naturally self-modifying.

Although the language has promising elements, it also has many limitations, which we discuss and try to overcome in this paper. First of all, we report our efforts in extending the language towards generic computations, beyond the original communication scope. After that, we show how a Fraglets program can operate on itself, benefiting from a flat code and data representation, and use this as a basis for implementing self-modification and self-replication. These two properties can be combined into self-reproduction and later evolution, however, this has not been achieved yet. Self-replication in a quine style is surprisingly easy. However, the self-modification procedure faces some difficulties, due to the characteristics of the Fraglets language. In spite of its simple syntax, which was designed for easy manipulation by automatic means, the semantics of operations is such that naive random transformations are most of the times lethal, although the resulting structures are always syntactically correct. We then propose a few fixes for problems encountered, and point to future directions on how to make the language realistic for full self-modification, and potentially later on for self-reproduction and evolution.

This paper is structured as follows: Section 2 introduces self-replicating, -modifying, -reproducing systems, and related chemical models. Section 3 briefly explains the Fraglets model and instruction set, highlighting some re-


*Bionetics* '07, December 10-13, 2007, Budapest, Hungary

cent language extensions. Section 4 shows how such self-replicating, -modifying, -reproducing programs can be built in Fraglets, from simple quines up to genetic operators. This exercise highlights the strong and weak points of the language in implementing such self- operations. We conclude the paper with some discussions on potential fixes and future research directions.

# 2. BACKGROUND AND RELATED WORK

The search for potential models of machines that can produce copies of themselves can be traced back to the late 1940's, with the pioneering work by John von Neumann on a theory of self-reproducing automata [33]. He described a universal constructor, a machine able to produce a copy of any other machine whose description is provided as input, including a copy of itself, when fed with its own description. Both the machine and the description are copied in the process, leading to a new machine that is also able to replicate in the same way.

The definition in [24] makes the distinction between replication and reproduction clear: Replication involves no variation mechanism, resulting in an exact duplicate of the parent entity; deviations from the original are regarded as errors. On the other hand, reproduction requires some form of variation, for instance in the form of genetic operators such as mutation and crossover, which may ultimately lead to improvement and evolution. These operators change the description of the machine to be copied, requiring a self-modification mechanism.

Replication, reproduction and variation in living beings are performed as chemical processes in the DNA. In the computer science context, they map therefore well to artificial chemical computing models, which attempt to mimic such processes in a simplified way. Numerous such artificial chemistries have been proposed [11], with the most varied purposes from studying the origins of life to modelling chemical pathways in cells, or simply providing inspiration for new, highly decentralized computing models.

In this section we discuss related research in models able to express replication, reproduction and variation in computer programs.

## 2.1 Self-Replicating Code

Since von Neumann set the basis for a mathematically rigorous study of self-replicating machines, many instances of such machines have been proposed and elaborated. An overview of the lineage of work in the area of self-replication can be found in [13, 23].

Self-replication is a special case of universal construction, where the input to the constructor (description) contains a description of itself. However, while universal construction is a sufficient condition for self-replication, it is not a necessary one. Indeed Langton [18] argued that natural systems are not equipped with a universal constructor. He relaxed the requirement that self-replicating structures must treat its stored information both as interpreted instructions and uninterpreted data. With this he showed that simple self-replicating structures based on dynamic loops instead of static tapes can be built. This spawned a new surge of research on such self-replicating structures [22, 23].

Most of the contributions to self-replication were done within the cellular automata (CA) framework, introduced by von Neumann. Self-replicating code was a later branch appearing in the 1960's, focusing on replication of textual computer programs. The work on self-replicating code was motivated by the desire to understand the fundamental information-processing principles and algorithms involved in self-replication, even independent of their physical realization.

The existence of self-replicating programs is a consequence of Kleene's second recursion theorem [16], which states, that for any program $p$ there exists a program $p'$, which generates its own encoding and passes it to $p$ along with the original input.

The simplest form of a self-replicating program is a *quine*, named after the philosopher and logician Willard van Orman Quine (1908-2000). A quine is a program that prints its own code. Quines exist for any programming language that is Turing complete. The *Quine Page* [30] provides a comprehensive list of such programs in various languages.

## 2.2 Self-Modifying Code

In a system that is required to constantly evolve and adapt, the ability to automatically modify or update its own code parts is essential. Since reliable and secure self-modification is still an open issue, self-modifying code has been banished from good practice software engineering.

However, self-modifying code plays a key role in Evolutionary Computation (EC) and Artificial Life (ALife), where evolution still occurs mostly in offline simulated worlds. In the case of EC, only the best programs which have been thoroughly tested via multiple fitness cases can be safely used. In the case of ALife, the main role of programs is simply to survive, and since they remain in a virtual world there is no risk for the end user.

Evolvable instruction set virtual machines are used in most well-known ALife systems, such as Tierra and Avida [20]. They resemble assembly language, which is easily self-modifiable: one can write on memory positions that include the own memory location of the code. This is used to evolve software that self-reproduces, adapts, seeks to survive, etc. A precursor of such machine language approach was Core Wars [7].

In *Ontogenetic Programming* [28] programs include self-modification instructions that enable them to change during the run. This was shown to be an advantage for adaptation to the environment [27].

The *Push* family of programming languages [25] is designed for a stack-based virtual machine in which code can be pushed onto a stack, where it can be manipulated as data, and later popped for execution. A variant of Push was used in *Autoconstructive Evolution* [26], where individuals take care of their own reproduction, and the reproduction mechanism itself can evolve (showing self-modification at the level of reproduction strategies). Recently [25], the self-modification aspect of the language has been enhanced by permitting the explicit manipulation of an execution stack, which has been shown to be an advantage in evolution.

## 2.3 Self-Reproducing Code

Since self-reproduction requires variation and thus self-modification, it must also include resilience to harmful replication errors in the form of a self-repair mechanism, or a selection mechanism able to detect and discard harmful code. Resilience is however not sufficient to ensure self-reproduction. The method must produce viable offspring

programs with high probability, i.e. programs that are syntactically correct and can also reproduce on their own. Otherwise the system spends most of its time trying to recover from bad mutations and does not optimize itself. The challenge is then how to design such viable self-reproducing schemes, or whether they could evolve from the interaction of simple molecules. The latter is associated to the origin of life problem which is intensively studied in ALife: reproduction was not a predesigned feature, it emerged out of chemical interactions in the primordial soup [12]. In complement to ALife systems, systems such as Autoconstructive Evolution [26] couple self-reproduction with a functional purpose in the algorithmic sense, beyond simple survival. This is also the line of research that we adopt.

## 2.4 Chemical Computing Models

Artificial chemical computing models [1, 5, 8, 21] express computations as chemical reactions that consume and produce objects (data or code). Objects are represented as elements in a *multiset*, an unordered set within which elements may occur more than once.

In [11] chemical computing models are classified as applications of Artificial Chemistry, a branch of Artificial Life (ALife) dedicated to the study of the chemical processes related to life and organizations in general. In the same way as ALife seeks to understand life by building artificial systems with simplified life-like properties, Artificial Chemistry builds simplified abstract chemical models that nevertheless exhibit properties that may lead to emergent phenomena, such as the spontaneous organization of molecules into self-maintaining structures [10, 12]. The applications of artificial chemistries go beyond ALife, reaching biology, information processing (in the form of natural and artificial chemical computing models) and evolutionary algorithms for optimization, among other domains.

Chemical models have also been used to express replication, reproduction and variation mechanisms [9, 11, 15, 29]. From these, we focus on models that apply these mechanisms to computer programs expressed in a chemical language, especially when these programs are represented as molecular chains of atoms that can operate on other molecular species, as opposed to models where a finite and well-known number of species interact according to predefined, static reaction rules. The interest of such molecular-chain models is two fold: first of all, complex computations can be expressed within molecule chains; second, they can more easily mimic the way in which DNA, RNA and enzymes direct reproduction, potentially leading to evolution in the long run.

Holland's Broadcast language [14] was one of the very earliest computing models resembling chemistry. It also had a unified code and data representation, in which broadcast units represented condition-action rules and signals for other units. They also had self-replication capacity, and the ability to detect the presence or absence of a given signal in the environment. The language seems to have never been implemented until recently [6], when it was shown to be helpful in modelling real biochemical signalling networks.

In [11], several so-called artificial polymer chemistries are described. In these systems, molecules are virtual polymers, long chains of "monomers" usually represented as letters. Polymers may concatenate with each other or suffer a cleavage at a given position. The focus of those models was to model real chemistries or to study the origin of life. More recently [31], a chemistry based on two-dimensional molecular chains (represented as strings of lines) has been proposed to model molecular computing, and has been shown to be able to emulate Turing machines.

Other chemistries based on strings that can represent code or data include [3, 9]. In [3] binary strings encode artificial regulatory networks able to perform complex functions. In [9] pairs of simple fixed-length binary strings react with each other: one of them represents the code and the other the data on which the code operates. The authors show the remarkable spontaneous emergence of a crossover operator after some generations of evolutionary runs.

We conjecture that a chemical language can express programs that can be more easily transformed and can become more robust to disruptions due to alternative execution paths enabled by a multiset model. Therefore they lend themselves more easily to self-modification, replication and reproduction. However, there are difficulties: the nondeterministic, decentralized and self-organizing nature of the computation model make it difficult for humans to control such chemical programs.

# 3. THE FRAGLET REACTION MODEL

A fraglet, or computation fragment, is a string of atoms (or symbols) `[`$s_1$ $s_2$ $s_3$ ... $s_n$`]` that can be interpreted as a code/data sequence, as a virtual "molecule" used in a "chemical reaction", or as a sequence of packet headers, or yet as an execution thread.

Fraglets are injected into a *virtual reaction vessel* where they are probabilistically selected for reaction, according to a well-stirred tank reactor algorithm. When selected, each fraglet only has its first (header) tag processed at a time. No deeper inspection within the fraglet is allowed. This ensure fairness across multiple execution threads represented by different fraglets in the reactor.

There are two types of reactions: *first-order reactions* or *transformations*, involving only one input molecule (reactant); and *second-order reactions* involving two reactants. The instruction set implementing the rules for such reactions will be described next.

## 3.1 Notation Conventions

Throughout this paper a number of conventions will be adopted to explain the language. These conventions are not part of the language itself, and are provided only to facilitate the explanations. An actual line of code in the Fraglets language is denoted by a line in the format `f [ ... ]`. Here is the set of conventions:

- Lower case words represent language keywords or other symbols appearing literally in the code.
- Single capital letters are wildcards for single atoms (e.g. `S`).
- Fully capitalized words of more than one letter (e.g. `T1, TAIL`) are placeholders for a sequence of zero or more atoms.
- Arithmetic expressions and conditions follow a C-like convention within parenthesis, e.g. ($cond$ ? $s1$ : $s2$) for "if $cond$ is true, then use $s1$ else use $s2$".

- The definition operator `::=` means that the left side expands into the right side; the left side is used in the text for conciseness, while the expanded form is the one that should appear in the actual code.

## 3.2 Basic Instruction Set

Table 1 shows the basic Fraglets instruction set, which includes the original instructions in [32] plus some new ones.

**Table 1: Fraglet reaction and transformation rules**

| *Reaction* | *Input* | *Output* |
|---|---|---|
| `match` | `[match S U T1]`, `[S T2]` | `[U T1 T2]` |
| `append` | `[append S U T1]`, `[S T2]` | `[U T2 T1]` |
| `matchp` | `[matchp S U T1]`, `[S T2]` | `[matchp S U T1]`, `[U T1 T2]` |
| *Transf.* | | |
| `dup` | `[dup U V TAIL]` | `[U V V TAIL]` |
| `exch` | `[exch U V W TAIL]` | `[U W V TAIL]` |
| `nop` | `[nop TAIL]` | `[TAIL]` |
| `nul` | `[nul TAIL]` | (fraglet is removed) |
| `pop` | `[pop U V H TAIL]` | `[U H]`, `[V TAIL]` |
| `split` | `[split U ...V * TAIL]` | `[U ...V]`,`[TAIL]` |

Transformations are standalone reactions: the operator keywords act on the fraglet itself. The general format of a transformation rule is: `[K U V W TAIL]` where `K` is the keyword for the operation; `U` is the output tag, the tag that will be the head of the resulting fraglet; `V` and `W` are input parameters to the operation; and `TAIL` represents the rest of the fraglet which is not processed by the current rule. For example, the transformation `exch` flips two input symbols, while `dup` takes one input symbol and duplicates it. The keyword is consumed in the process and the output tag `U` becomes the new head of the fraglet. Apart from `split` which splits one fraglet in two, all other transformations work alike.

The reaction rules basically merge the tails of two fraglets under the condition that their head tags match exactly. The difference between the original `match` and the new `append` instruction (introduced for convenience) is the order of the tails in the result. The `matchp` rule is a persistent version of `match` in which the rule itself is not consumed during the reaction (analogous to a catalyst in chemistry); this is useful for recurring code.

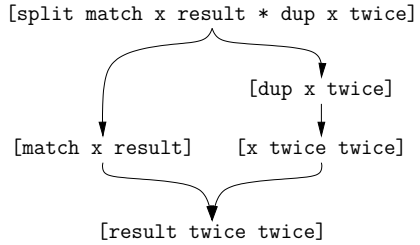


**Figure 1: Execution flow of a fraglet**

Figure 1 shows the execution flow of a fraglet which consists of several different instructions. The `split` instruction first split the fraglet into two parts. The resulting fraglets are processed independently, i.e. the `dup` instruction transforms one of the fraglets by duplicating its third atom. Finally, the `match` instruction defines a reaction rule that joins the execution flow by combining the two fraglets as described in table 1.

## 3.3 Arithmetic Operators and Conditionals

Fraglets were extended with basic arithmetic and conditional operators expressed as transformations as listed in Table 2.

**Table 2: Arithmetic Operators**

| *Keyword* | *Input* | *Output* |
|---|---|---|
| `sum` | `[sum R A B TAIL]` | `[R (A+B) TAIL]` |
| `eq` | `[eq Y N U V TAIL]` | `[((U==V) ? Y : N) U V TAIL]` |
| `lt` | `[lt Y N U V TAIL]` | `[((U<V) ? Y : N) U V TAIL]` |

Several arithmetic operators are implemented: `sum`, `sub`, `mult`, `div`, `mod`, `pow` (sum, subtraction, multiplication, division, modulo, power, respectively). For conciseness, Table 2 shows only the example of `sum`. All the other operators have a similar syntax.

These operators can be combined into arbitrary arithmetic expressions as in the simple example below:

```
f [match x match a mult t]  # t = a*x
f [match t match b sum fx]  # f = b+t
```

The code above implements the function $f(x) = ax + b$. The input parameters are expected in tags `a` and `b`, and the function argument in tag `x`. Tag `t` guards the intermediate result of $ax$. The result will be stored in a fraglet with head `fx`. As an example, when `[a 3]`, `[b 4]`, and `[x 10]` are provided then the result `[fx 34]` is obtained.

Two conditionals are supported, `eq` (equal) and `lt` (less than). They return the results of the comparisons into either of the two front tags: if the comparison is true, the first tag is used, otherwise the second. Example:

```
f [eq yes no water fire rest]
f [lt yes no 2 3 rest]
```

The first line will produce `[no water fire rest]` while the second line will produce `[yes 2 3 rest]`. Note that the compared atoms are not discarded, such that they can be used afterwards if necessary. This is useful since there is no explicit way to encode variables in Fraglets.

## 3.4 Supplementary Instructions

Some frequently used operations are implemented as instructions in Fraglets for convenience. These instructions could be implemented with the instruction set explained in previous sections (basic instructions, arithmetic, conditionals). However, they would require to iterate over the entire fraglet string, by tearing it apart and reconstructing it. Efficiency can be gained by implementing them directly in the interpreter.

**Table 3: Supplementary instructions**

| *Keyword* | *Input* | *Output* |
|---|---|---|
| `length` | `[length U TAIL]` | `[U` *length*`(TAIL) TAIL]` |
| `empty` | `[empty Y N TAIL]` | `[((TAIL == []) ? Y : N TAIL)]` |
| `fork` | `[fork U V TAIL]` | `[U TAIL], [V TAIL]` |

Table 3 shows these supplementary instructions. They behave as follows: `length` returns the length of a fraglet's tail in symbols. For instance, `[length l a b c]` returns `[l 3]`. `empty` checks whether the tail of a fraglet is empty, and if yes, returns the first tag; else it returns the second tag followed by its non-empty tail. `fork` duplicates an entire fraglet and prepends different head tags to them, such that they can be separately identified and processed independently.

# 4. SELF-GENERATION OF CODE

In this section we provide examples of self-replication and self-modification in the Fraglets framework. We use quines as templates for self-replicating programs, show how to embed useful functions into them, and how to attach self-modification properties to them. Then we provide ideas of how to design evolutionary operators like mutation and crossover.

## 4.1 Self-Replication

Quines are programs that produce their complete source code as output. On this note quines are ideal objects by which we can study self-replication properties.

In general, a quine consists of two parts, one which contains the executable code, and the other which contains the data. The data represents the blueprint of the code. The information that is stored in the blueprint is used twice during replication: First it serves as instructions to be interpreted by the quine to construct a new quine. Then the same information is attached to the new offspring, so that it is able to replicate in turn.

The usage of the information to build instructions can be compared to the *translation* of genes occurring in cells, where RNA chains carrying the genotype are translated into proteins. The latter use of the blueprint resembles the DNA *replication*.

### *Self-Replication in Fraglets*

In Fraglets it is easy to implement both *translation* and *replication* of data "molecules", since the Fraglets language has a flat code/data representation. The following example shows a reaction trace of a data fraglet `x` that is *translated* (interpreted as executable code) by a `match` rule.

`[match x],[x sum result 4 5]` $\longrightarrow$ `[sum result 4 5]`

This *translation* process in Fraglets is performed as an explicit removal of the passive header tag which activates the rule such that it can be executed. During execution, the active and the passive parts react together, resulting in a new fraglet that calculates the sum. Information *replication*, on the other hand, can be achieved by using a fork instruction. Hence a simple quine can be built by finding a code and a data fraglet that react and, in doing so, replicate both.

Here is a first example of a simple quine:

```
f [  match x fork nop x]
f [x match x fork nop x]
```

In this example, depicted in figure 2, two copies of the information are present: the first is the executable (active) copy, and the second is the code storage (blueprint), guarded by tag `x`. Their reaction produces a fraglet with `fork` instruction, which performs *translation* and *replication* of the information at the same time. One copy is executed again, restarting the cycle, and the other reinstalls the original blueprint.

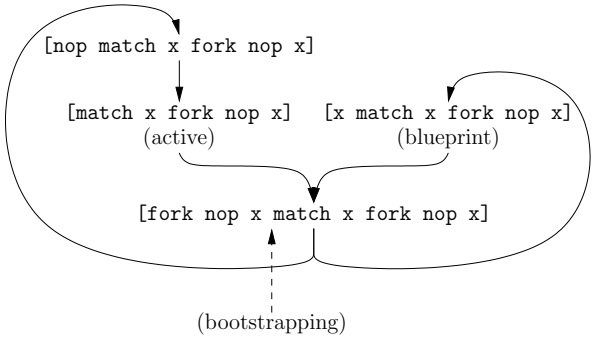


**Figure 2: Execution flow of a quine**

We observe that the active and the passive parts look similar. In fact, they only differ in their head symbol: the passive part is tagged with a tag `x`, whereas the active part directly starts with the `match` instruction. The following fraglet "bootstraps" the quine shown above:

```
f [fork nop x match x fork nop x]
```

The quine presented in this section is an example for simple self-replication. It does not do any useful computation and spends cycles only to regenerate itself.

### *Self-Replication with Embedded Functionality*

The next example can compute any function expressed as a `CONSUME` and `PRODUCE` part:

```
[  QUINE]
[x QUINE]
```

where:

```
QUINE     ::= CONSUME REPLICATE PRODUCE
REPLICATE ::= split match x fork nop x *
```

For example, to generate a quine replacement for the header rewriting catalyst `[matchp in out]`, we can define

```
CONSUME ::= match in
PRODUCE ::= out
```

which results in:

```
f [  match in split match x fork nop x * out]
f [x match in split match x fork nop x * out]
```

Like in the previous case, the information is copied during execution: one copy is *replicated* to the passive form `[x match in ...]` and the other is *translated* and executed. At the same time, the program performs its intended functionality, i.e. to rewrite the input tag to the output tag. Quines that perform more complex computations can be written by just specifying the production and consumption sides accordingly. The following example yields a quine that multiplies two numbers:

```
CONSUME ::= match in1 match in2
PRODUCE ::= mult out
```

One difference between a quine and the `[matchp ...]` rule is the late reproduction of the quine's code compared to the built-in catalyst. While the rule `[matchp in1 in2 mult out]` persists after consuming the first input `in1`, the corresponding quine replicates later, after consuming both inputs. Late reproduction is useful to prevent the rule from binding to multiple `in1` inputs without existing `in2` fraglets.

## 4.2 Self-Modification

The next stage is to build a self-modifying program. Starting with the previous example, the quine now uses a simple mutation operator that changes the production side during information *replication*. At the same time, a copy of the blueprints is made available as messenger blueprints for later translation into the active parts. This can be compared to the DNA to RNA *transcription* in cells, where RNA chains act as *messengers*, which are translated into proteins by ribosomes.

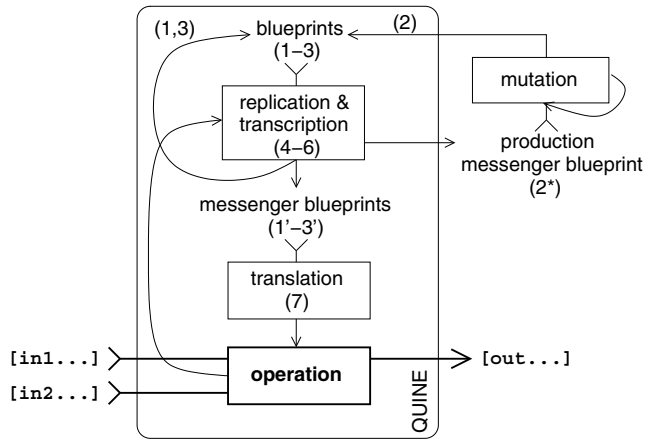


**Figure 3: Self-Modifying Quine**

Figure 3 depicts the principle of the self-modifying quine. The resulting program is more complex than the previous examples, because it needs to regenerate multiple fraglets:

```
1   [consume CONSUME]
2   [produce PRODUCE]
3   [replicate REPLICATE]
4   [REPLCONS]
5   [REPLPROD]
6   [REPLREPL]
7   [GENERATE]
```

where:

```
CONSUME   ::= match in1 match in2
PRODUCE   ::= mult out
REPLICATE ::= split REPLCONS * split REPLPROD *
              split REPLREPL * split GENERATE *
REPLCONS  ::= match consume fork qcons consume
REPLPROD  ::= match produce fork qprod mprod
REPLREPL  ::= match replicate fork qrepl replicate
GENERATE  ::= match qcons match qrepl match qprod
```

The mutation operator for the production messenger is:

```
8   f [matchp mprod split getop * match reop mprod2]
9   f [matchp mprod produce]9
10  f [matchp mprod2 pop mprod3]
11  f [matchp mprod3 exch produce]
12
13  f [matchp getop match op fork op resop]
14  f [op sum]     f [op diff]
15  f [op mult]    f [op div]
```

The three passive fraglets (1-3) contain the blueprints for the consumption and the production rules, as well as the blueprint for the code that replicates the remaining rules (4-7). The purpose of fraglets 4-6 is to *replicate* the blueprints (1-3), and to make a copy of them (1'-3') available to rule 7. Rule 7 plays the role of *translation* by building the actual operation. This operation consumes input data, produces output data, and then regenerates the quine's code.

The execution flow is as follows:

1. The active fraglets 4-6 immediately react with fraglets 1-3, respectively. Each reaction:
   (a) Consumes the respective blueprint.
   (b) Duplicates the embodied information. One copy becomes the blueprint again (*replication*), while the other copy becomes the messenger blueprint (*transcription* to 1'-3'), which is later consumed by rule 7.
2. However, fraglet 5 does not *replicate* the blueprint itself, but handles another copy (2*) to the mutation operator.
3. The mutation operator changes the arithmetic operation of the production messenger blueprint with a probability of 10 percent. This is done by having two competing rules that process the production messenger blueprint (2*). The first (8) performs the modification by picking an arithmetic operator (14,15) randomly, while the second (9) leaves the operator untouched. The relative concentrations of these two rules define the probability of mutation (in the example, 1 mutation rule against 9 non-mutation ones, leading to 10% mutation probability).
4. Rule 7 consumes the messenger blueprints (1'-3') generated by fraglets 4-6, and generates a new active fraglet (*translation*) that performs the following actions:
   (a) Consumes both input fraglets `in1` and `in2`.
   (b) Performs the arithmetic calculation specified by the production blueprint.
   (c) Regenerates the three *replication* and *transcription* fraglets 4-7 using the embedded messenger blueprint (3').

At the end, the whole quine has been reproduced, while the production side has potentially been modified.

In this example we used a controlled mutation, in which an arithmetic operator at a well-defined position was exchanged with another operator of the same arity. This allowed viable mutations. In the following section we show the issues involved in building general-purpose genetic operators like mutation and crossover in the Fraglets language.

## 4.3 Towards Self-Reproduction

The step from code variation by controlled mutation (as showed in the previous section) towards generic self-reproduction requires that a program undergoing mutation and crossover must be inherently robust to the application of these operators. In this section we focus on the implementation of mutation and crossover operators in Fraglets, which can manipulate long fraglet chains, as opposed to single operators as in the previous section. Naive operators lead to code disruption, but the construction of intelligent operators leading to viable code is an open issue.

### 4.3.1 Mutation

At first glance it appears to be fairly easy to mutate a fraglet as the set of operations and the reaction model offer the basic features required for this task.

The general idea is to copy the sequence of symbols of an input fraglet into its blueprint, symbol by symbol. Every time the head is split from the input fraglet we apply the same mutation principle as in our quine example in Subsection 4.2. There are multiple rules which compete for the head symbol to be copied. The concentration of these rules influences the probability with which they are going to process the selected symbol.

These rules have the same structure: they consume the head symbol and generate a new head with tag `newhead`. Consequentially, the simplest rule is:

```
f [matchp head newhead]
```

This rule is equivalent to rule 9 in Section 4.2, which preserves the fraglet. The higher its concentration the less likely the mutation of a symbol is. A low mutation rate requires a low concentration of the competing rules that actually mutate symbols. An example of such rule is:

```
f [matchp head split newhead x * nul]
```

This sample mutation rule deletes the head and generates the new symbol `x`, thus mutating the original symbol to `x`. Of course, `x` can be replaced by any other symbol, an instruction, or the result of a complex computation.

The fraglet with tag `newhead` is finally appended to the current blueprint. After consuming all symbols, the blueprint is a mutated version of the input fraglet.

Obviously, this mutation is uncontrolled: mutated symbols may yield fraglets which do not have coherent execution flows. This is one of the major problems which we have to face in this language. They are discussed in more depth in Section 5.

### 4.3.2 Crossover

A crossover operator exchanges genetic code between two parents and generates new offsprings which share code of both parents. For this purpose the parents are split at one or two point(s) (1/2-point crossover) and their front parts are exchanged.

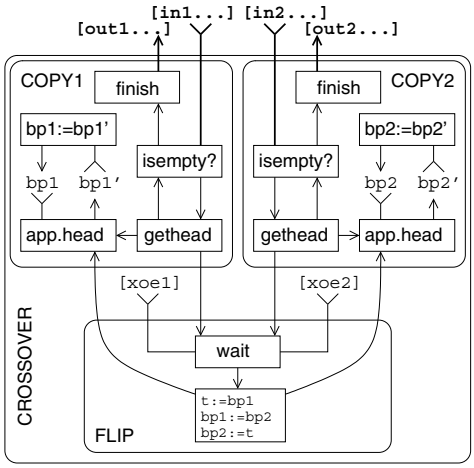


**Figure 4: Execution flow of crossover**

Figure 4 illustrates an abstraction of a possible implementation of the crossover operator. The rectangles represent *modules* which group fraglets associated with a specific task.

The crossover uses two instances of a copy module. Similar to the mutation described above, this module splits (`gethead`) the head `h` from the input fraglet one at a time. If the input fraglet is completely consumed, i.e. it is empty (`isempty?`), the result is passed to the output. The result is the blueprint (`bp`$x$) which is modified in each step by appending (`app.head`) head `h`.

At `gethead` we again use the same method as in the quine and mutation implementations. Several rules compete for symbol `h`. These are the rules in modules `app.head` and `wait`. If rules of the first module consume `h` the execution flow stays in the copy module. Otherwise the execution continues in module flip. Consequentially, the concentration of the fraglets in these modules also control the probability of a crossover during the copy process.

The flip module waits (`wait`) until the second copy module passes its head symbol to the flip module. If this happens, the actual crossover can take place. It exchanges the current blueprints, thus distributing parts of the parents to the new offsprings. Afterwards, the heads are passed back to the copy modules and the copying process proceeds.

Depending on the `match` instruction executed first, the execution flow will branch at random into the `copy` or the `flip` module. Without any restrictions on this process one may end up with a fraglet which consists of a random number of fragments from each input fraglet. Thus, to implement a 1-point crossover the branching of the execution flow to the flip module has to be controlled in some way. We do this by providing fraglets `[xoe1]` and `[xoe2]` as triggers for the matching rules in `wait`. One pair of them can trigger one crossover. In this way, we can also implement 2-point crossovers.

Similarly to the mutation operator we were able to show that a crossover can be implemented using Fraglets. However, this implementation suffers from the same problems. Although we are able to control the number of crossover points and despite the fact that we can implement matching

rules in `wait` that more intelligently pick a crossover point, it is not possible to guarantee that the generated offsprings possess a coherent execution flow. On the contrary, performing a crossover on two fraglets implies merging two execution flows which possibly use different symbols as matching tags and thus generating new fraglets which can not execute at all.

Hence, although we can show that Fraglets can easily implement basic genetic operators it becomes obvious that the invasive character of these operations requires more work to ensure that programs subject to these manipulations are still executable. The following section discusses some possibilities to achieve this.

# 5. DISCUSSION

Section 4.2 showed how simple and controlled mutations can lead to viable programs, however the scope of the transformations is limited to replacing an operator with another one with equivalent arity. Section 4.3 then showed how more generic genetic operators can be implemented in Fraglets, which could modify them at arbitrary places. However a prevalent problem with these intrusive manipulations of fraglets is the destruction of the execution flow.

A possible solution to be considered is the implementation of simple symbol rewriting instructions in Fraglets. They would have to analyze the execution flow of a fraglet. Based on this information they could detect possible disruptions of the execution flow and rewrite instructions or symbols accordingly. As dynamically generated fraglets may influence the execution, the analytical part of this process is very complicated. We have already identified simple heuristics which may be used to analyze whether a fraglet or a fraglet set, for example, produces the symbols it is going to consume and vice versa. Second, code which is able to analyze the structure of fraglets can also be used to improve the quality of genetic operators. For example, it could support a crossover which only exchanges code fragments which are located in similar contexts [19]. In this way we would already limit the destructive behaviour of the genetic operators. Similarly, we could perhaps use this structural details to support GP schemata [17].

A related problem is how to get rid of invalid or unused active reaction rules. A decay mechanism for these fraglets could be a possible solution. This would require an accompanying selection mechanism that would determine which rules maintain themselves via self-replication, and which ones decay. This in turn would require a resource control mechanism, possibly based on mass conservation, that would ensure that resources are distributed according to some fitness criterion. This would reflect nature where limited resources create competition between individuals, which drives selection and evolution.

# 6. CONCLUSIONS AND OUTLOOK

Our goal in this paper was to show how self-replicating and self-modifying programs can be created using Fraglets as a chemical programming language. Obtaining self-replication and self-modification properties is a first step towards self-reproducing programs. Self-reproduction is an important property for future autonomic systems. We described general methods to develop self-replication and -modification structures using Fraglets.

In particular we showed how Fraglets programs are able to replicate due to the flat code/data representation. Starting from simple quines, we elaborated on self-replicating programs of increasing complexity. The resulting programs are still able to execute the intended computation.

We also subjected self-replicating programs to simple and controlled mutations. The resulting mutants were viable yet modified offspring of their parents. We also showed how more generic variation operators, like mutation and crossover, can be expressed in Fraglets.

A full self-reproduction scheme is still not demonstrated due to the problem of obtaining general-purpose variation operators that produce viable Fraglets programs with a high probability. Currently, naive genetic operations disrupt the execution flow of Fraglets programs, which is heavily dependent upon matching tags to trigger reactions.

Our long-term objective is to create stable, yet self-healing programs that survive internal (mutations) and external (attacks) perturbations. To this end, genetic operators that can produce viable offspring automatically are needed. As a future work, we plan to examine dynamic properties of the Fraglets framework and to further investigate biochemical reaction networks, in order to stabilize execution paths within a Fraglets program.

# 7. ACKNOWLEDGMENTS

This work has been supported by the European Union and the Swiss National Science Foundation, through FET Project BIONETS and SNP Project Self-Healing Protocols, respectively. We would also like to thank Prof. Christian Tschudin for his valuable comments.

# 8. REFERENCES

[1] J.-P. Banâtre, P. Fradet, and Y. Radenac. A Generalized Higher-Order Chemical Computation Model with Infinite and Hybrid Multisets. In *1st International Workshop on New Developments in Computational Models (DCM'05)*, pages 5–14, 2005. To appear in ENTCS (Elsevier).

[2] J.-P. Banâtre, Y. Radenac, and P. Fradet. Chemical specification of autonomic systems. In *Proc. 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE'04)*, pages 72–79, July 2004.

[3] W. Banzhaf. Artificial Regulatory Networks and Genetic Programming. In *Genetic Programming - Theory and Applications, R. Riolo, B. Worzel (Eds.)*, chapter 4, pages 43–61. Kluwer Academic, Boston, MA, 2003.

[4] W. Banzhaf, P. Dittrich, and H. Rauhe. Emergent Computation by Catalytic Reactions. *Nanotechnology*, 7:307–314, 1996.

[5] C. S. Calude and G. Paun. *Computing with Cells and Atoms: An Introduction to Quantum, DNA and Membrane Computing*. Taylor & Francis, 2001.

[6] J. Decraene, G. G. Mitchell, B. McMullin, and C. Kelly. The Holland Broadcast Language and the Modeling of Biochemical Networks. In Ebner et al., editor, *Proceedings of the 10th European Conference on Genetic Programming (EuroGP 2007)*, volume 4445 of *LNCS*, pages 361–370, Valencia, Spain, Apr. 2007.

[7] A. K. Dewdney. Recreational Mathematics – Core Wars. *Scientific American*, May 1984. See also http://www.koth.org/ and http://www.corewars.org/.

[8] P. Dittrich. Chemical Computing. In *Unconventional Programming Paradigms (UPP 2004), Springer LNCS 3566*, pages 19–32, 2005.

[9] P. Dittrich and W. Banzhaf. Self-Evolution in a Constructive Binary String System. *Artificial Life*, 4(2):203–220, 1998.

[10] P. Dittrich and P. S. di Fenizio. Chemical organization theory: towards a theory of constructive dynamical systems. *Bulletin of Mathematical Biology*, 69(4):1199–1231, 2005.

[11] P. Dittrich, J. Ziegler, and W. Banzhaf. Artificial Chemistries – A Review. *Artificial Life*, 7(3):225–275, 2001.

[12] W. Fontana and L. W. Buss. The Arrival of the Fittest: Toward a Theory of Biological Organization. *Bulletin of Mathematical Biology*, 56:1–64, 1994.

[13] R. A. Freitas Jr. and R. C. Merkle. *Kinematic Self-Replicating Machines*. Landes Bioscience, Georgetown, TX, USA, 2004. available online http://www.molecularassembler.com/KSRM.htm.

[14] J. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1992. First Edition 1975.

[15] T. J. Hutton. Evolvable Self-Reproducing Cells in a Two-Dimensional Artificial Chemistry. *Artificial Life*, 13(1):11–30, Winter 2007.

[16] S. Kleene. On notation for ordinal numbers. *The Journal of Symbolic Logic*, 3:150–155, 1938.

[17] W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer, 2002.

[18] C. G. Langton. Self-reproduction in cellular automata. *Physica D*, 10D(1-2):135–144, 1984.

[19] H. Majeed and C. Ryan. A less destructive, context-aware crossover operator for GP. In P. Collet, M. Tomassini, M. Ebner, S. Gustafson, and A. Ekárt, editors, *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 36–48, Budapest, Hungary, Apr. 2006. Springer.

[20] C. Ofria and C. O. Wilke. Avida: A software platform for research in computational evolutionary biology. *Artificial Life*, 10(2):191–229, 2004.

[21] G. Paun. Computing with Membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.

[22] J.-Y. Perrier, M. Sipper, and J. Zahnd. Toward a Viable, Self-Reproducing Universal Computer. *Physica D*, 97:335–352, 1996.

[23] M. Sipper. Fifty years of research on self-replication: an overview. *Artificial Life*, 4(3):237–257, 1998.

[24] M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Perez-Uribe, and A. Stauffer. A Phylogenetic, Ontogenetic, and Epigenetic View of Bio-Inspired Hardware Systems. *IEEE Transactions on Evolutionary Computation*, 1(1), Apr. 1997.

[25] L. Spector, J. Klein, and M. Keijzer. The Push3 execution stack and the evolution of control. In *Proc. Genetic and Evolutionary Computation Conference (GECCO 2005)*, pages 1689–1696, 2005.

[26] L. Spector and A. Robinson. Genetic Programming and Autoconstructive Evolution with the Push Programming Language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, 2002.

[27] L. Spector and K. Stoffel. Automatic generation of adaptive programs. In P. Maes, M. J. Mataric, J.-A. Meyer, J. Pollack, and S. W. Wilson, editors, *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior: From animals to animats 4*, pages 476–483, Cape Code, USA, 9-13 Sept. 1996. MIT Press.

[28] L. Spector and K. Stoffel. Ontogenetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 394–399, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[29] C. Teuscher. From membranes to systems: self-configuration and self-replication in membrane systems. *BioSystems*, 87(2-3):101–110, Feb. 2007. from the Sixth International Workshop on Information Processing in Cells and Tissues (IPCAT 2005), York, UK, 2005.

[30] G. P. Thompson. The quine page. http://www.nyx.net/∼gthompso/quine.htm.

[31] K. Tominaga, T. Watanabe, K. Kobayashi, M. Nakamura, K. Kishi, and M. Kazuno. Modeling Molecular Computing Systems by an Artificial Chemistry—Its Expressive Power and Application. *Artificial Life*, 13(3):223–247, 2007.

[32] C. Tschudin. Fraglets – A Metabolistic Execution Model for Communication Protocols. In *Proc. 2nd Annual Symposium on Autonomous Intelligent Networks and Systems (AINS)*, Menlo Park, USA, July 2003.

[33] J. von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA, 1966.