Robustness to Code and Data Deletion in Autocatalytic Quines

Thomas Meyer[1], Daniel Schreckling[2], Christian Tschudin[1], and Lidia Yamamoto[1]

[1] Computer Science Department, University of Basel
Bernoullistrasse 16, CH–4056 Basel, Switzerland
{th.meyer,christian.tschudin,lidia.yamamoto}@unibas.ch
[2] Computer Science Department, University of Hamburg
Vogt-Koelln-Str. 30, D–22527 Hamburg, Germany
schreckling@informatik.uni-hamburg.de

Abstract. Software systems nowadays are becoming increasingly complex and vulnerable to all sorts of failures and attacks. There is a rising need for robust self-repairing systems able to restore full functionality in face of internal and external perturbations, including those that affect their own code base. However, it is difficult to achieve code self-repair with conventional programming models.

We propose and demonstrate a solution to this problem based on self-replicating programs in an artificial chemistry. In this model, execution proceeds by chemical reactions that modify virtual molecules carrying code and data. Self-repair is achieved by what we call *autocatalytic quines*: programs that permanently reproduce their own code base. The *concentration* of instructions reflects the health of the system, and is kept stable by the instructions themselves. We show how the chemistry of such programs enables them to withstand arbitrary amounts of random code and data deletion, without affecting the results of their computations.

1 Introduction

Today's computer programs are engineered by humans using the classical technique of building an abstract model of the real world and implementing reactions to this modelled world. Hence it is not surprising that these programs often fail in the real world environment, because their designers did not foresee certain situations, because of implementation errors or malicious attacks. In addition, as devices become smaller the influence of electical noise and cosmic radiation increases, resulting in transient faults that lower the reliablity of the calculation outcome [1]. Thus there is a need to build reliable systems based on unreliable components where the software is resilient against internal and external perturbations and recovers itself from such situations.

Self-recovery means adapting the internal model to maintain properties like correctness, effectiveness, etc. This generic kind of adaptation is not possible with pre-programmed strategies. Because most programming languages available

today separate code from data, a program once deployed cannot be changed in order to adapt to environmental changes. This lack of elasticity makes it hard to build programs that react to unpredicted situations. One approach to address the challenge of constructing self-healing software could be the use of self-modifying programs. However, there is still little theory about how to design such programs [2].

Our approach is to use self-replicating programs in an artificial chemistry. Artificial chemical computing models [3–5] express computations as chemical reactions that consume and produce data or code objects. In an artificial chemistry, computations can occur at microscopic and macroscopic scales [4]. At the microscopic scale, we observe how reactions operate on the information stored in individual molecule instances; at this scale, a change in a single molecule has a visible and immediate impact on the result of the computation. At the macroscopic scale, in contrast, molecules occur at massive numbers, and a change in a single molecule is unlikely to have a significant impact on the system as a whole; at this level, it is the concentration of different molecular species that mainly determines the outcome of the computation, which is ready when the system reaches a steady state [6, 7].

While artificial chemistries designed for biological modelling have focused on computation at macroscopic scales [7, 8], chemical computing abstractions targeted at concurrent information processing have traditionally focused on microscopic scales [9, 10]. We adopt a hybrid approach in which code molecules are able to self-replicate in order to maintain their concentration and survive deletion attacks. This results in a fault tolerance mechanism observable at the macroscopic scale, while computations are performed at the microscopic level. These self-replicating code molecules are called *autocatalytic quines*. Quines are programs that produce their own code as output, and autocatalytic refers to their ability to make copies of themselves, thus catalyse their own production.

We show how autocatalytic quines are able to perform some simple function computation while at the same time keeping their concentrations stable in the presence of a dilution flux, and also in the presence of targeted deletion attacks with a significant decrease in their molecule concentrations. The resulting fault tolerance is based on a constantly dynamic and regenerating system, as opposed to traditional techniques which seek to maintain the system as static as possible. All our experiments are performed using the *Fraglet* programming language [11, 12] in which self-replicating code can be programmed in a straightforward way. As a consequence, the characteristics of Fraglets and their reaction model will serve as a case study for a brief security analysis of artificial chemical computing models outlined above.

This paper is structured as follows: Section 2 introduces self-replicating systems and related chemical models. Section 3 briefly explains the Fraglet model and instruction set. Section 4 shows how such self-replicating sets of “molecules” can be built in Fraglets, from simple quines up to generic operators. Section 5 then analyzes the dynamic aspects of self-replicating sets. We show how to achieve robustness to deletion of molecules by imposing a dilution flow to the re-

action vessel that limits the growth of autocatalytic quines. In Sect. 6 we show a robust program that calculates an arithmetic expression using quine operations.

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 [13]. He described a universal constructor, a (mechanical) 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 [14] 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 [5], 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 self-replicating code and artificial chemistries.

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 [15, 16].

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 [17] argued that natural systems are not equipped with a universal constructor. He relaxed the requirement that self-replicating structures must treat their 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 [18, 16].

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 [19], 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* [20] provides a comprehensive list of such programs in various languages.

2.2 Artificial Chemistries

Artificial chemical computing models [10, 3, 4, 9] 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 [5] 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 [21, 22]. The applications of artificial chemistries reach 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 [23, 5, 24, 25]. 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 [26] 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. These units also had self-replication capacity, and the ability to detect the presence or absence of a given signal in the environment. The language was

recently implemented [27], and revealed helpful in modelling real biochemical signalling networks.

In [5], 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. In [23] 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. More recently [28], 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.

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 non-deterministic, decentralized and self-organizing nature of the computation model make it difficult for humans to control such chemical programs.

3 The Fraglet Reaction Model

Fraglet is an execution model inspired by chemistry originally aimed at the synthesis and evolution of communication protocols [11]. 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. There are a fixed number of production rules describing substitution pattern that operate on fraglets. We limit ourselves to substitution patterns which, on their left side, only depend on the first symbol of a word. For example, the rule `[exch S T U TAIL]` $\rightarrow$ `[S U T TAIL]` when applied to the word `[exch a b c d]` will result in `[a c b d]` – that is, two symbols are swapped. The `exch` acted as a prefix command for the rest of the word whereas the new left-most symbol '`a`' serves as a continuation pointer for further processing of the result. Table 1 shows some of the production rules.

According to [5], an artificial chemistry can be broadly defined by a triple (S, R, A), where S is the set of molecular species, R is the set of collision (reaction) rules, and A is the algorithm for the reaction vessel where the molecules S interact via the rules R. In the case of Fraglets, the set of molecules S consists of all possible fraglets. (The term molecule and fraglet is used synonymously in this paper.) The finite set of production rules implicitly define the potentially infinite set of reactions R among molecules.

The dynamic behaviour of a simulation is characterized by the algorithm A. Fraglets are injected into a virtual reaction vessel, which maintains a multiset of

Table 1. Selected production rules of a Fraglet system. `S, T` and `U` are placeholders for symbols, `TAIL` stands for a potentially empty word of symbols.

Instruction	Educt(s)		Product(s)
`exch`	`[exch S T U TAIL]`	→	`[S U T TAIL]`
`send`	$_{n_i}$`[send n`$_j$` TAIL]`	→	$_{n_j}$`[TAIL]`
`split`	`[split PART1 * PART2]`	→	`[PART1]` + `[PART2]`
`fork`	`[fork S T TAIL]`	→	`[S TAIL]` + `[T TAIL]`
`sum`	`[sum S i`$_1$` i`$_2$` TAIL]`	→	`[S `i_1+i_2` TAIL]` (do. for mult etc)
`match`	`[match S TAIL1]` + `[S TAIL2]`	→	`[TAIL1 TAIL2]`
`matchp`	`[matchp S TAIL1]` + `[S TAIL2]`	→	`[matchp S TAIL1]` + `[TAIL1 TAIL2]`

fraglets and simulates their reactions using the Gillespie algorithm [29]. The algorithm calculates the collision probability of two molecules in a well-stirred tank reactor. The reaction rate is proportional to the concentration of the reaction educts.

Fraglet programs normally consist of a set of active `match` or `matchp` fraglets that process passive (data) molecules. In order to make such programs robust to the deletion of constituting parts we aim at constantly replicate the `match` fraglets while they are being executed. In the next sections we show how self-replication can be realized using the Fraglet reaction model. In Sect. 4 we examine how to replicate the information stored in a set of fraglets. To analyze these static aspects of self-replication we focus on the set of molecules and reaction rules (S, R). Then, in Sect. 5 we include the dynamic behavior into our considerations and therefore discuss the reaction algorithm A in more detail.

4 Self-Replication in Fralgets

In this section we provide examples of self-replication in the Fraglet framework. Our goal is to find a set of fraglets that is able to maintain itself, i.e. to replicate all fraglets that are part of this set. We use quines as templates for self-replicating programs, and show how to embed useful functions into them.

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 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.*

In Fraglets it is easy to implement both *translation* and *replication* of data molecules, since the Fraglet language has a flat code/data representation. The following example shows a reaction trace of a data fraglet `bp` that is *translated*

(interpreted as executable code) by a `match` rule.

`[match bp] + [bp sum y 4 5]` $\longrightarrow$ `[sum y 4 5]` $\longrightarrow$ `[y 9]`

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. The "collision" of the active `match` and the passive fraglet cause the active and the passive parts react together, resulting in a new active 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, regenerate both.

Here is an example of a simple self-replicating, autocatalytic quine program:

```
[   match bp fork fork fork nop bp]
[bp match bp fork fork fork nop bp]
```

In this example, depicted in Fig. 1, 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 `bp`. Their reaction produces a fraglet starting with `[fork fork ...]` which *replicates* the information by generating two copies of the remaining part. The resulting two identical fraglets again start with a `fork` instruction. They individually *translate* the information: One copy reduces to the active part, restarting the cycle, and the other reinstalls the original blueprint.

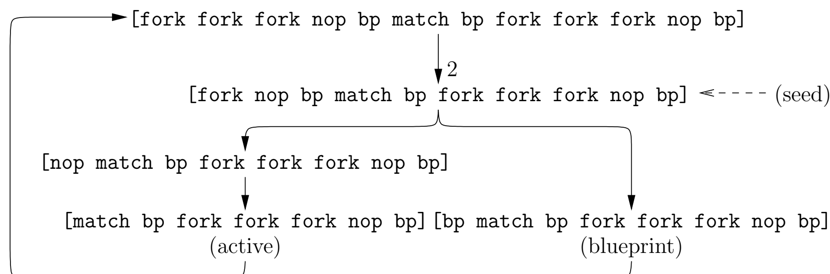


Fig. 1. Simple autocatalytic quine

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 `bp`, whereas the active part directly starts with the `match` instruction. Note that a single "seed" fraglet is sufficient to generate (bootstrap) the quine.

4.1 Embedding Functionality

The quine presented before is an example for simple self-replication. It does not do any useful computation and spends cycles only to replicate itself. Some

related work also concentrate on the structure of such self-replicating sets of molecules: Kaufmann, Bagley, Farmer, and later Mossel and Steel examined the properties of random autocatalytic sets [30–33], Fontana [22] as well as Speroni di Fenizio [34] studied the formation of organizations in a random reaction vessel of λ-expressions and combinator algebra, respectively. Most of them concentrate on the structure of self-replicating sets. Here we aim at performing microscopic computation [4], where the computation is carried out on instances of input molecules and the result is stored to an instance of an output molecule. A simple example for such microscopic computation is the following Fraglet reaction that computes the expression $y = x + 1$:

`[match x sum y 1]` + `[x 5]` ⟶ `[sum y 1 5]` ⟶ `[y 6]`

where the active `match` molecule reacts with a passive (data) molecule `x` that carries the input data. The resulting fraglet `sum` increments the number stored within itself and immediately produces a passive output molecule `y` containing the result.

Consider the following quine template, which can compute any function expressed as a *consume* and *produce* part:

[*spawn consume replicate produce*]

where generally

spawn ::= `fork nop bp`
replicate ::= `split match bp fork fork` *spawn* `*`

For example, to generate a quine replacement for `[match x sum y 1]`, we can define

consume ::= `match x`
produce ::= `sum y 1`

which results in the following quine *seed* code:

```
[fork nop bp match x split match bp
fork fork fork nop bp * sum y 1]
```

The reaction graph for this quine is shown in Fig. 2. We can distinguish three interdependent pathways: the upper cycle replicates the blueprint, the cycle in the middle generates the active rule (`[match x ...]`), and the lower pathway consumes an input molecule `x`, performs the actual computation and produces an output molecule `y`. Such as for the simple quine, the information of the decorated quine itself is copied during execution: one copy becomes the passive form `[bp match x ...]` and the other is executed. However, the decorated quine can

only replicate when an input molecule `x` is available. At the same time, the program performs its intended functionality, i.e. to calculate the expression. Quines that perform more complex computations can be written by just specifying the production and consumption sides accordingly.

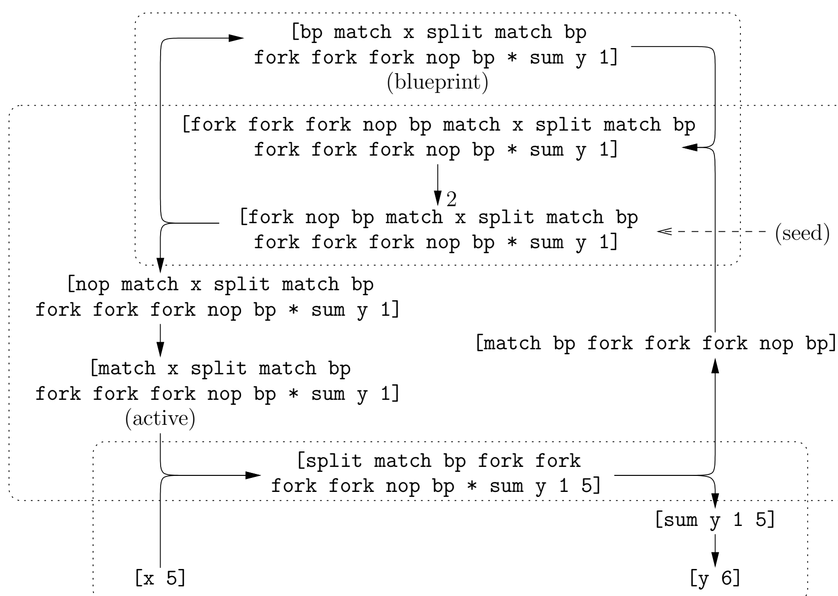


Fig. 2. Autocatalytic quine with embedded functionality

Operation. The decorated quine is an ideal building block to create larger programs. In this paper we refer to such a building block as an *operation*. The graphical short notation for the operation that calculates $y = x + 1$ is depicted in Fig. 3. It hides the replication details and only reveals the performed calculation.

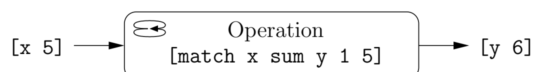


Fig. 3. Operation = Decorated autocatalytic quine (short notation)

5 Dynamic Aspects of Self-Replication

In this section we examine the dynamic aspects of the Fraglet algorithm in more detail. First we show two dynamic counterforces, growth and dilution flow that when acting together exhibit the desired property of robustness to code and data deletion.

5.1 Growth

So far we only analyzed the reaction network of the quines from the functional point of view. Let us turn back to the simple quine in Fig. 1 and analyze its dynamic behavior. In each round, the active and the passive part of the quine react and generate two copies of themselves. Each of the two resulting quine instances again reacts and yield two other instances. In a real chemical reaction vessel with constant volume, the collision probability of the reaction partners would rise. Such behavior can be simulated using the Gillespie algorithm [29], which makes sure that the simulated (virtual) time evolves by smaller increments the more possible reactions can take place in parallel. Since the possible reaction partners grow with the number of quine instances, the virtual time increments continuously shrink. As a result the quine population grows exponentially with respect to virtual time: the quine catalyzes its own replication.

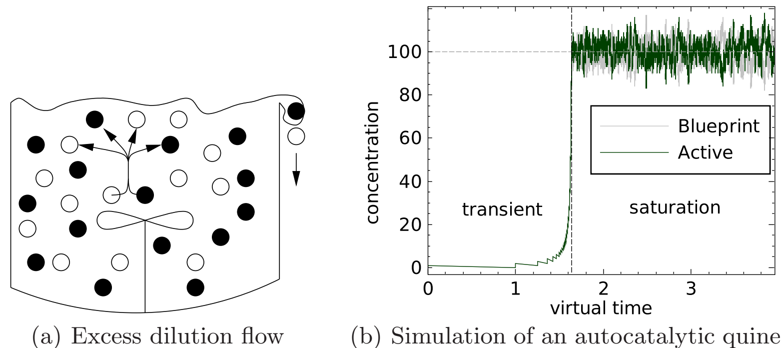
For the decorated quine (operation) shown in Fig. 2 we already noted that it only replicates itself when consuming an input molecule `x`. Thus the replication rate of an operation depends on the rate at which input molecules are injected into the reaction vessel. When input molecules appear at a constant interval, the operation exhibits a linear growth.

5.2 Dilution

The simulation of a growing population of quines on a computer quickly leads to a situation where the virtual time increment of a simulated reaction is smaller than the real time the computer requires to perform the reaction and to calculate the next iteration of the algorithm. In this case the computer cannot perform a real-time simulation of the reaction system anymore which is crucial for programs interfacing a real world environment like programs for robot control or network protocols.

Thus we limit the maximum number of molecules in the reaction vessel to a certain maximum vessel capacity N. For this purpose we apply a random excess dilution flow, which is (1) non-selective, i.e. it randomly picks one of the molecule instances for dilution using a uniform probability distribution, and where (2) the rate of the dilution flow depends on the overall production rate of the system.

Figure 4(a) shows the operating principle of the excess dilution flow. The reaction vessel is in either of two states: in the transient state the number of molecules is smaller than the specified limit and the dilution flow is inactive, whereas in the saturation state the number of molecules is equal to the specified limit and the dilution flow is active. Whenever a reaction produces new molecules



(a) Excess dilution flow (b) Simulation of an autocatalytic quine

Fig. 4. Excess dilution flow and the time evolution of a simulation of an autocatalytic (exponentially growing) quine using the Gillespie algorithm with a max. vessel capacity of $N = 200$.

in the saturation state, the excess dilution flow randomly selects and destroys molecules until the vessel reaches the max. fill level N. The excess dilution flow is able to keep the overall number of molecules at this level even though the population of a molecule set like the quine is growing exponentially.

A similar method has already been used by others, for example in [22, 34]. Formally, the system can be described by the *network replicator equation* [35].

Figure 4(b) shows the dynamic behavior of a simple autocatalytic quine in a reaction vessel limited by an excess dilution flow. In the transient state the quine population grows exponentially with respect to virtual time. When reaching the specified vessel capacity N each newly produced molecule replaces another randomly picked molecule.

Since the dilution flow is a random process, it may temporarily favor either the blueprint or the active fraglet of the quine resulting in a higher relative concentration of the other fraglet. However, fraglets with a higher concentration are more likely being selected by the dilution flow in the next iteration of the algorithm. On the long run both the active and passive fraglets reach the same concentration of $\frac{N}{2} = 100$.

5.3 Emergent Robustness to Code Deletion

One of the emergent properties of an exponentially growing population limited in a finite environment is robustness to the loss of individuals. An unexpected loss of any quine molecule instance decreases the total number of molecules, but this “hole” is immediately filled by the offsprings of the remaining quine instances.

Figure 5 depicts the result of a data and a code deletion attack. At virtual time $t_v = 3$ we remove 80 % of the blueprints. As the replication of the remaining autocatalytic quines continues they quickly replenish the missing molecules. Immediately after the attack the reaction vessel is in the transient state again: The remaining quine instances are able to replicate without being diluted, and

therefore the population grows exponentially. When reentering the saturation state, there are temporarily more active rules than blueprints. But the restored dilution flow causes the active fraglets being diluted more frequently. As soon as the equilibrium between blueprints and active rules is reobtained the system reaches its original steady-state and continues operating as before the attack. The same holds for an attack to the active fraglets, shown in Fig. 5 at $t_v = 4$.

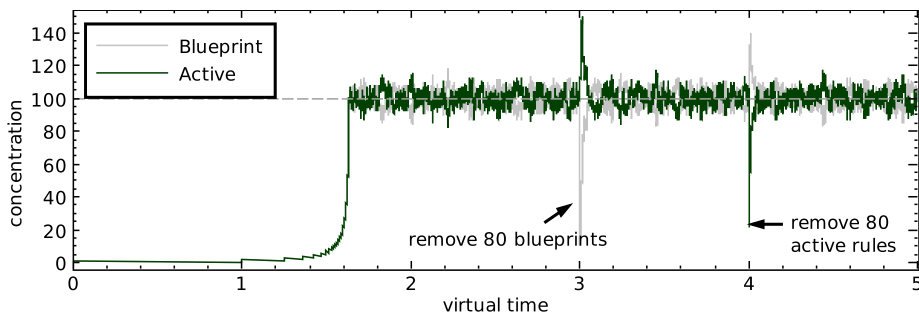


Fig. 5. Code and data deletion attacks. Time evolution of the simulation of an autocatalytic quine with deletion attacks at $t_v = 3$ (blueprints) and $t_v = 4$ (active fraglets) using the Gillespie algorithm with a max. vessel capacity of $N = 200$

6 Robust Programs Using Self-Replicating Operations

In this section we focus on how to build programs out of autocatalytic quines (operations). The goal is to arbitrarily combine "quine-protected operations" such that the overall resulting program is still robust to code and data deletions.

6.1 Example: Arithmetic Expression

As an example we build a program that calculates the arithmetic expression $y = 3x^2+2x+1$. Since Fraglets only has binary (arity 2) arithmetic operators we have to split up the calculation into different operations. Each operation consumes a certain input molecule and produces an intermediate result molecule that is then consumed by another operation. Figure 6 shows the reaction flow of the resulting program using 7 operations. The first operation forks the input fraglet `x` into two intermediate data fraglets `t1a`, and `t1b`, respectively. Then further calculation is performed in parallel by two chains. The left chain calculates the square root of the input and multiplies it with 3 whereas the right chain calculates the partial sum of $2x + 1$. Finally, the last instruction joins the resulting data fraglets of the two chains and calculates the sum over the partial results.

Figure 7 shows the result of a simulation focusing on the quine concentrations of the splitting operation 1 and the joining operation 7. The system quickly

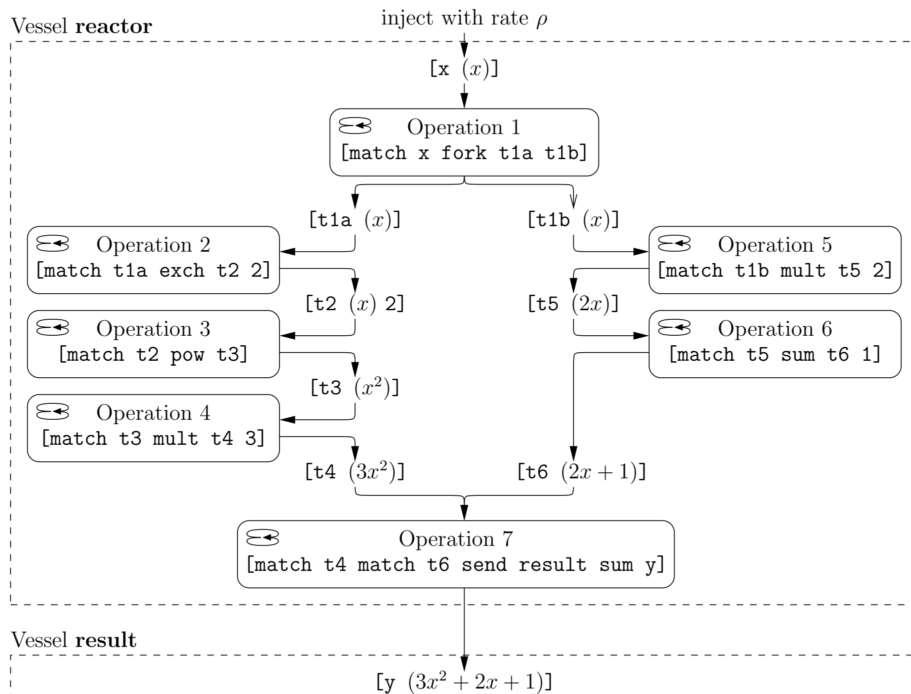
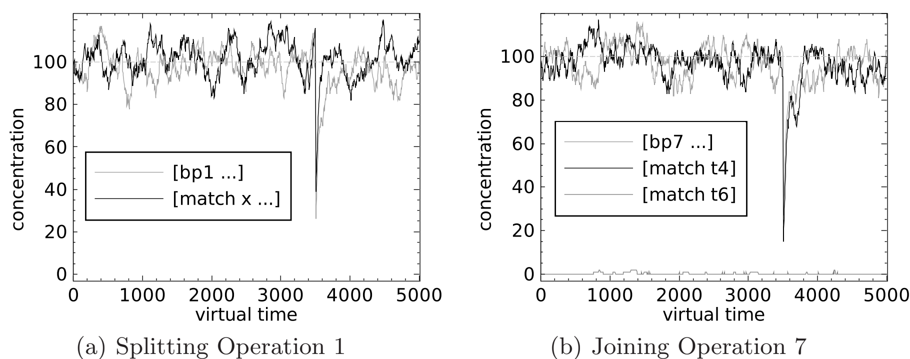


Fig. 6. Parallel chains of catalytic quines (operations) calculating the arithmetic expression $y = 3x^2 + 2x + 1$



(a) Splitting Operation 1 (b) Joining Operation 7

Fig. 7. Time evolution of the simulation of a partially parallel program to calculate the arithmetic expression $y = 3x^2 + 2x + 1$ using autocatalytic quines (operations). The Gillespie algorithm drives the system, and an excess dilution flow with a maximum vessel capacity of $N = 1400$ restricts unlimited growth. One input molecules x is injected with rate $\rho = 1$. At $t_v = 3500$, 80 % of all molecules in the reaction vessel are destroyed.

recovers from a deletion attack where 80 % of all molecules are removed and returns to its equilibrium where the N molecules are evenly distributed among blueprints and active fraglets of all operations.

6.2 Discussion

We showed how programs can be build by sequentially assembling decorated autocatalytic quines. The simulated burst deletion attacks already gave an indication that the overall system is robust to the deletion of molecules. Here we will further examine the dynamic properties of the resulting system.

Qualitative Considerations. A program is only able to survive in the reaction vessel if all its operations process the same data stream or if all processed data stream rates are in the same order of magnitude: We already observed that the rate of the data stream determines the replication rate of an operation. Because of the dilution flow and the operations competing for concentration, its replication rate also define the steady-state concentration of the operation's blueprint and active fraglets. For example, if the replication rate of one operation is twice the replication rate of another, the latter one will be present only with half of the concentration of the first and thus, the probability that it will become extinct because of the random fluctuation of the algorithm rises. In our example all operations replicate with more or less the same rate, because all operations process the same data stream and without loops in one of the parallel branches.

To learn more about the dynamic aspects and characteristics of the overall program we now further analyze the system by changing its input parameters and examining the resulting behavior.

Three input parameters have an impact on the behavior of a given program: (P.1) The max. vessel capacity N is held constant during the simulation. (P.2) The input molecules are actually generated by an unknown process outside the reaction vessel. Here we assume a constant injection rate ρ. (P.3) The rate and shape of the molecule deletion attack is the third input parameter. Unlike in our previous simulations we now assume a constant rate δ at which molecules are randomly picked and destroyed.

The behavior of the system can be characterized by the following three metrics: (M.1) The robustness of the overall system, measured by the probability that the system "survives" a simulation run; (M.2) data yield, expressed as the fraction of injected data molecules `x` that are not lost, i.e. that are converted to an output molecule `y`; (M.3) the required CPU power to simulate the system for a given set of input parameters.

Figure 8 qualitatively depicts the influence of input parameters $(P.1) - (P.3)$ to the system metrics $(M.1) - (M.3)$. Each input parameter is assigned an axis of the three-dimensional space, and an arbitrary point in the input parameter space is marked with a dot. The arrows that are leaving the dot indicate an increase of the labled metrics in consequence of changing one of the input parameters. For example, the arrow to the right shows that an increase of the data injection rate ρ requires more CPU power to simulate the system.

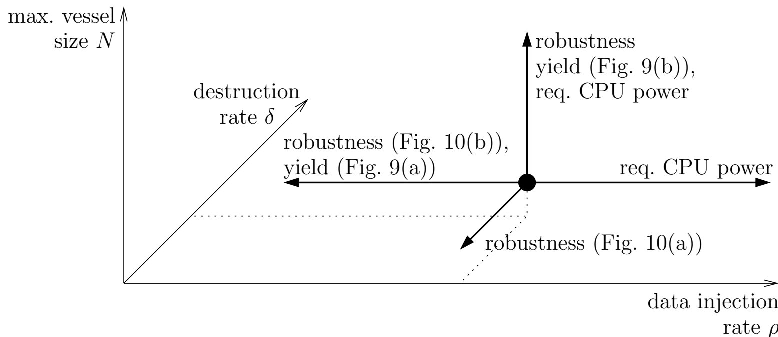


Fig. 8. Qualitative influence of the parameters $(P.1) - (P.3)$ to the metrics $(M.1) - (M.3)$. The figures refered to in parenthesis show their quantitative relation.

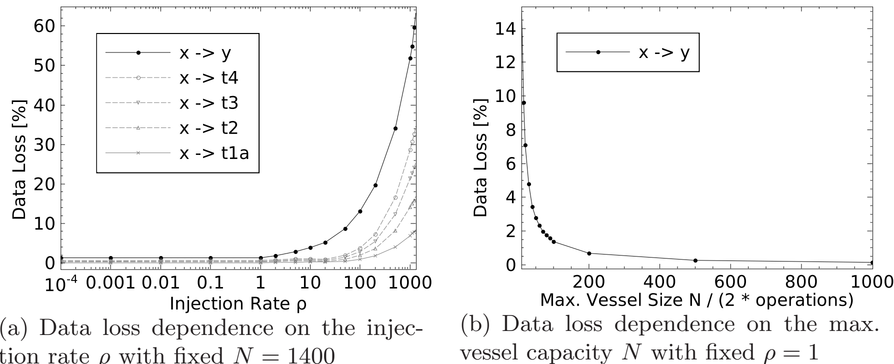
Qualitatively, we expect that the robustness of the system and the yield of the data stream can be increased by rising the max. vessel capacity N or by decreasing the rate ρ of injecting input molecules if possible. The robustness is obviously higher if the deletion rate δ is low. More CPU power is needed for higher injection rates or bigger reaction vessels. In the following we show quantitative measures for these tendencies, obtained by repeated simulation runs.

Quantitative Considerations. All the results discussed below are obtained by taking the average of 20 simulation runs. For each simulation we inject 10000 input molecules at rate ρ into the reaction vessel of max. capacity N and apply a constant deletion rate δ.

Data Loss. Before focusing on the robustness measure we analyze the yield, i.e. how many of the injected molecules are processed by all operations and finally reach the result vessel. We want to keep the data loss as low as possible, but we expect that even without a deletion attack the excess dilution flow also removes some of the molecules belonging to the processed data stream.

The yield is measured by comparing the average rate at which result molecules `y` are produced (ρ_y) to the rate at which input molecules `x` are injected (ρ): $yield = \frac{\rho_y}{\rho}$. Figure 9(a) shows the dependency of the data loss on the data injection rate. The concentration of data molecules is very low compared to the concentration of the operation's blueprint and active molecules. Hence data molecules are less frequently selected by the dilution mechanism and therefore, for moderate injection rates ρ, the data loss is surprisingly low despite the excess dilution flow.

When we increase the data injection rate ρ, one instance of the first operation may still be in progress of replicating itself after having processed the previous data molecule. Thus the concentration of its active rules is slightly lower and the reaction rate of the first operation drops. Like this a larger amount of input molecules may accumulate waiting for being processed by the first operation and



(a) Data loss dependence on the injection rate ρ with fixed $N = 1400$

(b) Data loss dependence on the max. vessel capacity N with fixed $\rho = 1$

Fig. 9. The fraction of data molecules that are lost between the injection of `x` and an intermediate or final product depends on the injection rate ρ and the max. vessel capacity N. (Here instead of N we use the expected target concentration of blueprint and active molecules for each operation: $\frac{N}{2o}$ where $o = 7$ is the number of operations.)

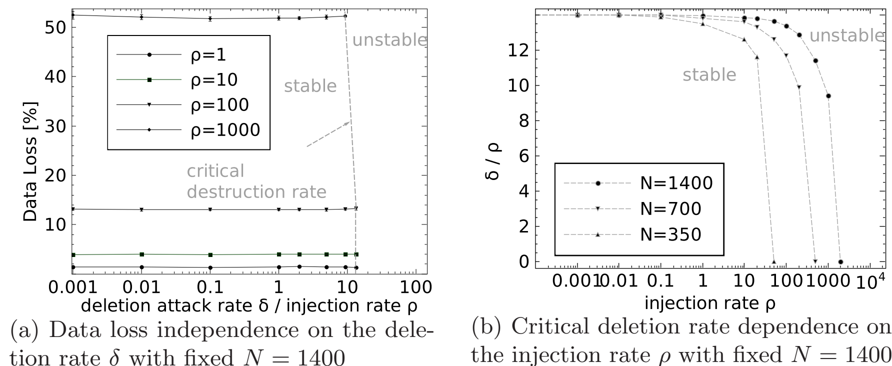
the probability that the input molecule is removed by the excess dilution flow rises, too. We can also see in Fig. 9(a) that each sequential operation imposes a certain data loss.

We can attenuate the influence of the injection rate ρ to the data yield by increasing the rate at which an operation processes the data stream. This can be done by increasing the concentration of the operation's active molecules, which can be obtained by increasing the max. vessel capacity N. Figure 9(b) shows how an increase of the max. vessel capacity N reduces the overall data loss for a certain data injection rate $\rho = 1$.

Surprisingly, the third input parameter, the rate of deletion attacks, has no effect on the data loss at all as shown in Fig. 10(a) for different data injection rates ρ. However, in this figure we can see that the system becomes unstable when the attack rate rises above a critical level. Hence we now show how the robustness of the system is influenced by the input parameters.

Robustness. A program shall be called robust if the probability to "survive" a simulation (where 10000 input molecules are injected with the given parameters) is greater than $P_{crit} = 0.9$. In other words, the system is robust if less than 10 % of all simulation runs encounter a starvation situation where no more result molecules `y` are generated when injecting input data molecules. Starvation might occur when one of the auxiliary molecules necessary for an operation becomes extinct, for example if the number of blueprints drops to zero.

Figure 10(b) depicts the critical attack rate beyond which the system becomes starved. For low data injection rates ρ the system is able to recover from a deletion attack δ that is about $2o$ times higher than ρ, where $o = 7$ is the number of operations. When increasing the injection rate, the average number of surviving systems quickly drop to zero. For a lower max. vessel capacity this critical injection rate is much lower.



(a) Data loss independence on the deletion rate δ with fixed $N = 1400$

(b) Critical deletion rate dependence on the injection rate ρ with fixed $N = 1400$

Fig. 10. Simulation of a constant random molecule deletion attack with rate δ. The fraction of data molecules that are lost between the injection of `x` and the final product `y` does not depend on the deletion rate δ. The critical deletion rate δ_{crit} beyond which the system becomes unstable depends on injection rate.

Required CPU Power. The CPU power required to execute a program is proportional to the injection rate ρ: The higher the rate of input molecules `x` the more instructions the algorithm must perform per time interval. In contrast the required CPU power only marginally depend on the vessel capacity N when keeping ρ constant. This is because the replication rate of the quines solely depend on the rate of the data stream. A small effect can be observed because an increase of N causes a cutback of the data loss and thereby more data molecules are able to pass all quines. Therefore when increasing N the number of algorithm cycles spent for each injected molecule only rises slightly.

General Observations. The quantitative analysis of the program showed that the input parameters must be chosen within certain margins in order to obtain a robust program. In the probabilistic execution model of an artificial chemistry one never reaches 100 % guarantees for a certain property.

The combination of autocatalytic quines and dilution flow has another stabilizing side effect: If invalid or useless code or data molecules are injected into the reaction vessel, they will eventually be removed by the dilution flow. Therefore the system exhibits some robustness to useless or harmful code. However, this assumption only holds if the injected molecules do not contain self-replicating molecules and if they don't interfere with the operations, as we discuss in following section.

7 From Robustness towards Security

So far we focused on the robustness of Fraglet programs to code and data deletion. In this section we extend the scope and consider further unintentional and malicious attacks to the code and data base.

The exploitation of code has a long history [36]. Over the years many techniques have been developed to exploit code vulnerabilities of common programming mistakes. Thus, these exploits mainly focus on the weaknesses of the code instead of exploiting the theoretical model of the language or the system it runs on. In artificial chemical computing systems the situation is different. Since code is executed via reactions among molecules, it is possible to modify existing molecules such that their computation corresponds to a different functionality. This is based on a characteristic that is deliberately left out of traditional programming languages such as C or Java: Any type of data, input or output data, also represents code and can thus have direct effect on the executed code.

Competing Molecules. If an attacker is allowed to arbitrarily inject molecules into the reaction vessel, he or she can alter the outcome of the computation and destory the integrity of the program in various ways. For example, the logic of the program presented in Sect. 6 can be changed by injecting competing molecules: The attacker could inject malicious code fraglets (`[match t3 mult t4 6]`) or data fraglets (`[t3 0]`), respectively. The first molecule competes against the active fraglet of operation 4 for the input data molecule `t3` and masks operation 4 in Fig. 6 to compute $y = 6x^2 + 2x + 1$. The molecule in the latter example directly modifies the result of an operation. However, both attacks are only of a temporary nature because the replicating quines soon reinstall the proper molecules.

Viral Blueprints. A more efficient attack targets the blueprints. An injected viral blueprint `[match bp1 bp1 ...]` replaces one of the blueprint instances for the first operation. As long as the concentration of the original blueprint is higher than the viral blueprint the latter will be displaced. But if the attacker injects a large number of modified blueprints the system will activate and replicate the viral blueprint.

Targeted Deletion Attacks. An attack can not only expoit existing code but also disrupt the program. Beyond the analyzed non-selective (blind) deletion attack we can also imagine a targeted deletion attack where an attacker selectively deletes molecules. Even if the attacker has no possibility to directly remove molecules from the reaction vessel he/she can inject active molecules to consume vital passive molecules. For example, the injected active molecule `[match bp1]` would consume and destroy a blueprint molecule needed to replicate the first operation. Similarly, one could inject molecules (`[match t2]`) to destroy intermediary results. In turn, all operations that depend on the suppressed data molecule cannot replicate anymore. If either attack is executed with a high number of molecules it drives the system to starvation.

Self-Optimization. Up to now, this section discussed intentional attacks which assumed an active attacker pursuing some specific goal and designing malicious code accordingly. However, the motivation for using artificial chemical computing

systems were also based on the fact that these languages possess self-* capabilities, including self-optimization. A self-optimizing Fraglet program could adapt itself to the environment by appling evolutionary methods, which in turn calls for random modification of its operations. We are therefore facing another threat of unintentional attacks, because random modification can in principle lead to code that performs any of the attacks discussed above.

Concluding Remarks. Our approach of using autocatalytic quines in an artificial chemistry concentrates on a special threat: the non-selective deletion of code and data molecules, which is one of the events that are likely to occur when operating on unreliable devices. We highlighted that there are a vast number of other threats that need to be adressed in future, but we also showed that some of the attacks can already be defended if the number of injected harmful molecules is below a certain limit.

A generic protection mechanism for mobile, self-modifying and self-reproducing code is naturally a big challenge beyond current knowledge. A hypothetical protection mechanism in this context would inherently need to be itself mobile, self-modifying and self-reproducing, such that it could track and respond to such dynamic attack patterns in an effective way. This could lead to an "arms race" of attack and protection waves. The challenge would be to stabilize such a system, in the manner of a biological immune system.

8 Conclusions

In this paper we demonstrated that within Fraglets as an artificial chemistry, it is possible to build self-maintaining structures that, when exposed to an open but resource constrained environment, exhibit robustness to deletion of their constituent parts. These properties were then brought forward from simple autocatalytic quines to programs consisting of autocatalytic building blocks. Detailed qualitative and quantitative analysis of the program's behaviour showed that the robustness to code and data deletion can be granted within certain input parameter margins.

The presented self-healing mechanism emerges from the combination of a growing population of autocatalytic quines and a random non-selective dilution flow. As a result the code of the program constantly rewrites itself and the dilution flow makes sure that non-replicating instructions vanish.

9 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.

References

1. Baumann, R.C.: Soft errors in advanced semiconductor devices – part 1: the three radiation sources. IEEE Transactions on Device and Materials Reliability **1**(1) (2001) 17–22
2. Anckaert, B., Madou, M., de Bosschere, K.: A model for self-modifying code. In Camenisch, J., ed.: Proceedings of the 8th Information Hiding Conference. Number 4437, Berlin Heidelberg, Springer-Verlag (2007) 232–248
3. Calude, C.S., Păun, G.: Computing with Cells and Atoms: An Introduction to Quantum, DNA and Membrane Computing. Taylor & Francis (2001)
4. Dittrich, P.: Chemical Computing. In: Unconventional Programming Paradigms (UPP 2004), Springer LNCS 3566. (2005) 19–32
5. Dittrich, P., Ziegler, J., Banzhaf, W.: Artificial Chemistries – A Review. Artificial Life **7**(3) (2001) 225–275
6. Banzhaf, W., Lasarczyk, C.: Genetic Programming of an Algorithmic Chemistry. In: Genetic Programming Theory and Practice II, O'Reilly et al. (Eds.). Volume 8. Kluwer/Springer (2004) 175–190
7. Deckard, A., Sauro, H.M.: Preliminary Studies on the In Silico Evolution of Biochemical Networks. ChemBioChem **5**(10) (2004) 1423–1431
8. Leier, A., Kuo, P.D., Banzhaf, W., Burrage, K.: Evolving Noisy Oscillatory Dynamics in Genetic Regulatory Networks. In: Proc. 9th European Conference on Genetic Programming. P. Collet, M. Tomassini, M. Ebner, S. Gustafson, A. Ekárt (Eds.) Springer LNCS 3905, Budapest, Hungary (2006) 290–299
9. Păun, G.: Computing with Membranes. Journal of Computer and System Sciences **61**(1) (2000) 108–143
10. Banâtre, J.P., Fradet, P., Radenac, Y.: A Generalized Higher-Order Chemical Computation Model with Infinite and Hybrid Multisets. In: 1st International Workshop on New Developments in Computational Models (DCM'05). (2005) 5–14 To appear in ENTCS (Elsevier).
11. Tschudin, C.: Fraglets - a metabolistic execution model for communication protocols. In: Proc. 2nd Annual Symposium on Autonomous Intelligent Networks and Systems (AINS), Menlo Park, USA (2003)
12. Yamamoto, L., Schreckling, D., Meyer, T.: Self-Replicating and Self-Modifying Programs in Fraglets. In: Proc. 2nd International Conference on Bio-Inspired Models of Network, Information, and Computing Systems (BIONETICS 2007), Budapest, Hungary (2007)
13. von Neumann, J.: Theory of Self-Reproducing Automata. University of Illinois Press, Champaign, IL, USA (1966)
14. Sipper, M., Sanchez, E., Mange, D., Tomassini, M., Perez-Uribe, A., Stauffer, A.: A Phylogenetic, Ontogenetic, and Epigenetic View of Bio-Inspired Hardware Systems. IEEE Transactions on Evolutionary Computation **1**(1) (1997)
15. Freitas Jr., R.A., Merkle, R.C.: Kinematic Self-Replicating Machines. Landes Bioscience, Georgetown, TX, USA (2004)
16. Sipper, M.: Fifty years of research on self-replication: an overview. Artificial Life **4**(3) (1998) 237–257
17. Langton, C.G.: Self-reproduction in cellular automata. Physica D **10D**(1-2) (1984) 135–144
18. Perrier, J.Y., Sipper, M., Zahnd, J.: Toward a Viable, Self-Reproducing Universal Computer. Physica D **97** (1996) 335–352

19. Kleene, S.: On notation for ordinal numbers. The Journal of Symbolic Logic **3** (1938) 150–155
20. Thompson, G.P.: The quine page. http://www.nyx.net/~gthompso/quine.htm (1999)
21. Dittrich, P., di Fenizio, P.S.: Chemical organization theory: towards a theory of constructive dynamical systems. Bulletin of Mathematical Biology **69**(4) (2005) 1199–1231
22. Fontana, W., Buss, L.W.: The Arrival of the Fittest: Toward a Theory of Biological Organization. Bulletin of Mathematical Biology **56** (1994) 1–64
23. Dittrich, P., Banzhaf, W.: Self-Evolution in a Constructive Binary String System. Artificial Life **4**(2) (1998) 203–220
24. Hutton, T.J.: Evolvable Self-Reproducing Cells in a Two-Dimensional Artificial Chemistry. Artificial Life **13**(1) (2007) 11–30
25. Teuscher, C.: From membranes to systems: self-configuration and self-replication in membrane systems. BioSystems **87**(2-3) (2007) 101–110 from the Sixth International Workshop on Information Processing in Cells and Tissues (IPCAT 2005), York, UK, 2005.
26. Holland, J.: Adaptation in Natural and Artificial Systems. MIT Press (1992) First Edition 1975.
27. Decraene, J., Mitchell, G.G., McMullin, B., Kelly, C.: The Holland Broadcast Language and the Modeling of Biochemical Networks. In Ebner et al., ed.: Proceedings of the 10th European Conference on Genetic Programming (EuroGP 2007). Volume 4445 of LNCS., Valencia, Spain (2007) 361–370
28. Tominaga, K., Watanabe, T., Kobayashi, K., Nakamura, M., Kishi, K., Kazuno, M.: Modeling Molecular Computing Systems by an Artificial Chemistry—Its Expressive Power and Application. Artificial Life **13**(3) (2007) 223–247
29. Gillespie, D.T.: Exact Stochastic Simulation of Coupled Chemical Reactions. Journal of Physical Chemistry **81**(25) (1977) 2340–2361
30. Bagley, R.J., Farmer, J.D., Kauffman, S.A., Packard, N.H., Perelson, A.S., Stadnyk, I.M.: Modeling adaptive biological systems. Biosystems **23** (1989) 113–138
31. Farmer, J.D., Kauffman, S.A., Packard, N.H.: Autocatalytic replication of polymers. Physica D **2**(1-3) (1986) 50–67
32. Kauffman, S.A.: The Origins of Order: Self-Organization and Selection in Evolution. Oxford University Press (1993)
33. Mossel, E., Steel, M.: Random biochemical networks: the probability of self-sustaining autocatalysis. Journal of Theoretical Biology **233**(3) (2005) 327–336
34. di Fenizio, P.S., Banzhaf, W.: A less abstract artificial chemistry. In Bedau, M.A., Mccaskill, J.S., Packard, N.H., Rasmusseen, S., eds.: Artificial Life VII, Cambridge, Massachusetts 02142, MIT Press (2000) 49–53
35. Stadler, P.F., Fontana, W., Miller, J.H.: Random catalytic reaction networks. Physica D **63**(3-4) (1993) 378–392
36. Hoglund, G., Mcgraw, G.: Exploiting Software: How to Break Code. Addison-Wesley Professional (2004)