Competition and Cooperation of Self-Healing Software

Thomas Meyer and Christian Tschudin

Technical Report CS-2010-004
University of Basel
Aug 3st, 2010

Abstract

Living systems reached a good balance between competition and cooperation in order to prevail – as individual or as a group. Artificial systems either work in isolation or are manually designed to cooperate, which is of paramount importance in networking applications. Recently, research considered bio-inspired approaches to increase the robustness of distributed algorithms. However, when mimicking natural rules such as applying natural selection, the resulting systems often compete rather than cooperate in the struggle for existence.

We recently presented an execution model for networking protocols inspired by chemical reactions in which we organized networking software as self-rewriting sets of “molecules”. If memory is limited, our protocol software exhibits remarkable robustness to faults and is able to run on unreliable hardware, because healthy software is able to replicate while faulty elements die out.

In this report we study the competitive nature of this environment and propose a methodology to design complex self-healing software that is able to cooperate therein. We resort to the study of self-organization in nature and adapt concepts like Eigen’s Hypercycle to our software. As an application case, we demonstrate how the competitive and cooperative forces can be exploited for a controlled update of software in a network.

Keywords: robustness, competition, cooperation, code replication, artificial chemistry, unconventional computing.
1 Introduction

Natural systems achieved a remarkable level of complexity. Nevertheless, and unlike traditional human artifacts, they nicely degrade in the presence of faults and are able to recover after a perturbation, exhibiting a high motivation to heal themselves. This is seen as the consequence of evolution through natural selection that promotes robustness and homeostatic behavior. Homeostasis is the phenomenological mechanism through which a system acts to maintain a stable internal environment despite external variations (Shaw, 2002).

Researches recently tried to adopt similar mechanisms for artificial systems after they recognized that the inevitable increase in complexity comes along with enhanced brittleness. Designs that enable software to heal itself from faults would radically improve its reliability and consistency (Ghosh, Sharman, Rao, & Upadhyaya, 2007). Artificial Immune Systems (AIS) (Timmis, Knight, De Castro, & Hart, 2004) is only one approach among many to achieve a higher level of autonomic robustness.

Our own work in this area resulted in an execution model for networking software that mimics chemical reaction networks (Tschudin, 2003; Meyer & Tschudin, 2009). Data packets as well as packet processing code are represented as virtual “molecules”; the distinction between code and data gets blurred. We let code continuously replicate itself in limited memory, potentially overwriting other code, resulting in a population of code strands that is highly robust to execution errors (Meyer & Tschudin, 2010).

In fact, code becomes too robust in this setting: once it populates the memory, it can barely be replaced. Furthermore, different software elements often struggle for the same memory resources and compete against each other rather than collaborate. A similar behavior was observed and became famous in Core War (Dewdney, 1984), where hostile programs engage in a battle of memory bits. Thus, the adoption of natural mechanisms comes along with natural forces, which we have to tame. We propose several methods how selfish code can be turned into symbiotic software that still exhibits the desired high level of robustness.

After a having highlighted the context of our approach and related work in Section 2, we proceed in Section 3 with introducing the “style” of implementing network services that we call chemical networking protocols. Section 4 presents self-replicating code strands and shows their robustness to execution errors. This self-healing code pattern is then extended in Section 5 in order to perform useful data operations. Section 6 reveals that uncorrelated self-healing code competes against each other whereas in Section 7 we demonstrate how to couple code parts such that complex software built from self-healing parts cooperates in order to fulfill a common goal. An application case (Section 8) makes use of the competitive as well as the cooperative forces to update software in a network. Finally, Section 9 concludes this report.

2 Context and Related Work

In this section, we reference the relevant corner stones for our work where we could draw important insights, namely fault tolerance, self-reproduction, artificial chemistries and their dynamics, and the dynamics of competing populations. The programming language “Fraglets”, which we have used to implement our system, was described by Tschudin (2003) and is summarized in Section 3.2.
2.1 Fault Tolerance

In computer science and engineering, the method of choice to achieve robustness, resilience and fault tolerance of services is to build up redundancy in order to mask errors (Johnson, 1996; Pradhan, 1996; Wilfredo, 2000): Multiple identical or similar redundant systems are performing the same task. The result is compared, often by a centralized observer. This architecture inevitably leads to the problem that the central decision maker may also be error prone, requiring an observer of the observer, and so on, leading to infinite regression. Hence, we need a solution where the central observer that steers the redundancy is redundant too and is blended into the system.

2.2 Self-Replication

Natural systems are able to dynamically construct redundancy by assembling and reproducing their components. Often, components exist in several copies (flocks, but also blood or nerve cells), exploiting parallelism and minimizing the impact of the loss of a single item. For singular components (e.g. bones) and in order to fight the problem of aging, redundancy is achieved over time through procreation, yielding a new and possibly modified copy. In computer science however, software is considered to be static (and without wear).

This view is recent: Back in the 1940’s, von Neumann (1966) developed a theory of self-reproducing automata. He described a universal constructor, a machine able to produce a copy of any other machine whose soft- and hardware blueprint is provided as input. Being universal, the constructor is also able to generate a copy of itself. Considerable research on self-replication was carried out on the framework of Cellular Automata (CA), in which remarkable results were achieved, also in terms of robustness and self-repair (Tempesti, Mange, & Stauffer, 1998). However, these results are hard to transfer from CA to the world of today’s computer software.

In the 1960s, with the desire to understand the fundamental information-processing principles and algorithms involved in self-replication, researchers started to focus on self-replicating code, how textual computer programs are able to replicate independent from their physical realization. The existence of self-replicating programs is a consequence of Kleene’s second recursion theorem (1938), 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, and made popular by Hofstadter (1979): A Quine is a program that prints its own code. Quines exist for any programming language that is Turing complete and it is a common challenge for students to come up with a Quine in their language of choice. The Quine Page provides a comprehensive list of such programs in various languages (Thompson, 2010).

In our work we put Quines in a parallel execution environment, permitting an ensemble of Quine copies to achieve surprising robustness with respect to code and packet loss and even execution errors. Our contribution consists in the demonstration of an operational system based on Quines that runs highly reliable network services with provable dynamic properties. More precisely, we will base on a chemical execution model, in which we place carefully crafted self-replicating programs. Packets, or “molecules”, react with each other and produce new packets, thus executing the program. Useful computations are piggybacked to the Quine structures in order to implement the network services.
2.3 Artificial Chemistry

Artificial chemical computing models (Banâtre, Fradet, & Radenac, 2006; Calude & Paûn, 2001; Dittrich, 2005; Holland, 1992; Paûn, 2000) express computations as chemical reactions that consume and produce objects (data or code). Objects are represented as elements in a multiset, which is an unordered set within which elements may occur more than once.

Dittrich, Ziegler, and Banzhaf (2001) classified chemical computing 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 (Dittrich & Speroni di Fenizio, 2007; Fontana & Buss, 1994).

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 (Dittrich & Banzhaf, 1998; Dittrich et al., 2001; Hutton, 2002; Teuscher, 2007; Yamamoto, Schreckling, & Meyer, 2007).

2.4 Chemical Kinetics

The dynamics of natural chemical reactions is governed by the law of mass action (Abrash, 1986) which states that the reaction rate is proportional to the reactant concentration. Several algorithms have been proposed to simulate chemical reactions on the microscopic level: Gillespie (1977) described an exact stochastic simulation algorithm that accurately mimics the randomness of reactions, which stems from the Brownian motion of the colliding molecules. Several variants and improvements of this algorithm have been proposed since then (Gibson & Bruck, 2000; Gillespie, 2007).

Originally, the aim of these algorithms was to simulate real chemical reactions. In this work, we use them as scheduling algorithms for our chemical programs. As a consequence, program execution is an inherently stochastic process; there is no guarantee, which reaction will be executed next and when. However, since on the macroscopic time scale these algorithms simulate the law of mass action, the average dynamic behavior can be described by the same Ordinary Differential Equations (ODEs) that are used to deterministically approximate real chemical reactions.

2.5 Population Dynamics

A novel element of our work is the use of hard limits to an artificial chemical vessel’s capacity. Environments with limited resources that host replicating entities lead to natural selection, as was shown by research on population dynamics by Fernando and Rowe (2007), Stadler, Fontana, and Miller (1993), Szathmáry (1991). In our case, the population consists of software components: Healthy software survives whereas errors are displaced. This naturally leads to software homeostasis – the intrinsic self-regulation of code in order to maintain a stable, healthy state.
In computer network research, mechanisms to control redundancy on the level of data-packets are well-known. Transmission control protocols such as TCP (Postel, 1981) are able to recover from packet losses. Within our setting, we are able to transpose this methods, currently only used for data stream control, down to the code execution level, granting resilience to the loss of parts of the protocol’s own software due to faults.

We will show that natural selection inevitably leads to a competitive environment where software components fight for resources and where this struggle may lead to the extinction of healthy but inefficient or rarely used code. The question is how to force the components to cooperate. Nature solved this problem at different levels: In higher organisms, altruism evolved, explained by Hamilton’s kinship theory (Hamilton, 1964). Wagner (2000) showed that signaling strategies evolve when individuals have to achieve a common task. This is, components have to communicate in order to cooperate. Even for lower level mechanisms, for example abstracted by game theory, a cooperative strategy may be beneficial for surviving (Axelrod & Hamilton, 1981; Axelrod, 1997). Eigen and Schuster (1979) presented a cooperative linkage of self-replicating chemical substances – the Hypercycle – an explanation of self-organization of prebiotic systems. We will use such hypercyclic structures in our artificial chemistry and further develop this idea for our execution model.

3 Chemical Networking Protocols

Traditionally, protocol execution is handled by a state machine that upon the reception of a packet synchronously changes its internal state and performs some communication activity. Here we introduce a “molecule metaphor” where each packet is treated as a virtual molecule. Virtual molecules react with other molecules in a reaction vessel (node). A reaction may produce other molecules being delivered to the application or being sent over the network. In such a chemical perspective, we obtain a web of reactions that together perform a distributed computation (called network service).

3.1 Modeling Chemical Communication

Instead of encoding a deterministic state machine, or having a sequential program that processes an incoming packet, each network node contains a finite multiset $M(S)$ of molecules $S = \{s_1, \ldots, s_n\}$ (=packets). In addition, each node defines a set of reaction rules $R = \{r_1, \ldots, r_m\}$ expressing which reactant molecules can collide and which molecules are generated during this process. Such a reaction is typically represented as

$$C_i + X_i \rightarrow C_i + X_j$$

The above reaction in node $i$ consumes, if present, two molecules $C$ and $X$ from the local multiset, regenerates $C$ and sends molecule $X$ to neighbor node $j$. In a simple two-node network topology, the above example spans the following reaction network, also depicted in Figure 1

$$C_1 + X_1 \xrightarrow{r_1} C_1 + X_2$$

$$C_2 + X_2 \xrightarrow{r_2} C_2 + X_1$$

---

T. Meyer, C. Tschudin

Competition and Cooperation
Figure 1: Distributed chemical reaction network: Each node provides a reaction that consumes two molecules C and X from the local multiset, regenerates C and sends molecule X to the neighbor node. The local reaction rules (2a) and (2b) span a global reaction network.

that works as follows:

A received molecule is in a first step passively placed into the multiset of the node. For example, rule (2b) is not executed immediately after node 2 receives a new X-molecule. It is rather scheduled for a later time determined by an exact stochastic reaction algorithm, for instance those proposed by Gillespie (1977) or Gibson and Bruck (2000). The reaction scheduler draws the delay between two occurrences of the same reaction from an exponential probability distribution. The role of this delayed execution is to enforce the law of mass action at the macroscopic level: Molecules C₁ and X₁ react with an average rate equal to the product of their abundance \( r₁ = c₁x₁ \). The rate of packets sent from node 1 to node 2 is equal to \( r₁ \) while the packet stream in the opposite direction exhibits a rate of \( r₂ = c₂x₂ \). It can be shown that the overall reaction system spanned by the two local reaction rules strives towards equilibrium where the number of X-molecules in either node is inversely proportional to the number of the corresponding C-molecules (Meyer & Tschudin, 2009).

3.2 Fraglets — a Chemical Programming Language

So far we demonstrated a static reaction networks where abstract reaction rules were “installed” permanently in each node. Here, we extend this model aiming at dynamically changing the set of reaction rules. We present the Fraglets language (Tschudin, 2003), an artificial chemistry according to the definition of Dittrich et al. (2001), whose corresponding chemical machine is executable and which serves as a simple platform to run chemical protocols.

Each molecule \( s \in \{S\} \), or packet, is a string of symbols over a finite alphabet \( \Sigma \). The first symbol of the string defines the string rewriting operation applied to this molecule by the virtual chemical machine; it can be thought of an assembler instruction. For example, the molecule \([\text{fork } a \ b \ c \ d] \) transforms itself and splits into the two molecules \([a \ c \ d] \) and \([b \ c \ d] \). The list below shows some essential instructions and their actions.

\[
\begin{align*}
\text{match } \alpha \Phi + [\alpha \Omega] &\rightarrow [\Phi \Omega] \\
[\text{fork } \alpha \beta \Omega] &\rightarrow [\alpha \Omega] + [\beta \Omega] \\
[\text{nop } \Omega] &\rightarrow [\Omega] \\
[\text{send } k \Omega] &\rightarrow [k\Omega] \quad \text{if } (i,k) \in \mathcal{E}
\end{align*}
\]

\( \alpha, \beta \in \Sigma \) are arbitrary symbols, \( \Phi, \Omega \in \Sigma^* \) are symbol strings, \( i,k \in \mathcal{V} \) denote network nodes and \( (i,k) \in \mathcal{E} \) communication links of the network graph \( \mathcal{G} = (\mathcal{V}, \mathcal{E}) \). Molecules starting with
match or any non-instruction identifier are in their normal form. The match-instruction can be used to join two molecules by concatenating the second to the first after removing the processed headers. Subsequent instructions immediately reduce the product further until they again reach their normal form. For example, the two molecules [[match pkt send 2 pkt]] and [[pkt data]] in node 1 imply the reaction

\[
[[\text{match pkt send 2 pkt}]] + [[\text{pkt data}]] 
\rightarrow [[\text{send 2 pkt data}]] 
\text{(transmission to neighbor 2)} 
\rightarrow [[\text{pkt data}]]
\]

Such a chemical language allows us to “program” the reaction graph. Molecules now have a structure; they contain information such as piggybacked user data. However, the dynamics of the reaction network is still governed by the law of mass action and thus, the protocol’s behavior is chemically controlled.

4 The Chemical Quine

In ordinary sequential programming languages, a “Quine” is a single piece of code outputting its own source code. In the parallel world of an artificial chemistry like Fraglets, a Quine becomes a set of molecules that is able to regenerate itself. An example that illustrates this concept is the combination of a blueprint molecule B = [[B fork nop match B]] and its active variant A = [[match B fork nop match B]] (Yamamoto et al., 2007). The two molecules react with each other and, according to the Fraglets rewriting rules, regenerate themselves as shown in Figure 2. The schematic illustration in Figure 2(b) shows the corresponding chemical reaction network that is dynamically equivalent to the Fraglets rewriting loop in Figure 2(a). Note that only bimolecular reactions are scheduled according to the law of mass actions; unimolecular rewriting rules such as fork are immediately executed, hence these intermediate steps (molecules) are omitted in the schematic notation.

![Rewriting loop in Fraglets](image1)

*a* Rewriting loop in Fraglets

![Reactions](image2)

*b* Reactions

**Figure 2: Chemical Quine in Fraglets:** A set of molecules (here: Fraglets strings) that regenerates itself. The blueprint molecule B reacts with its active variant A. The consecutive rewriting steps regenerate the two molecules.

4.1 Replicating Quine and Limited Resources

By repeating the fork instruction three times, the above Quine can be converted into a “replicating Quine” as shown in Figure 3. The replicating Quine generates two copies of itself in each round while consuming the original copy. Because the reactions are scheduled according to the
law of mass action, the overall production rate increases with the growing number of Quine instances. Consequently, the population of Quines grows hyperbolically (Szathmáry, 1991), meaning that it theoretically reaches infinite abundance in finite time.

As a limit to this unbounded growth we introduce a non-selective dilution flux to the reaction vessel, which destroys arbitrary molecules as long as the total number of molecules exceeds a pre-defined vessel capacity. This leads to a selective pressure: Only molecules that are part of a self-replicating set have a chance to remain present – all other molecules will eventually be displaced.

![Diagram](image)

**Figure 3: Replicating Quine**: The replicating Quines increases its population by generating two replicas while the original copy is consumed.

### 4.2 Dynamic Behavior and Deterministic Fixed Points

The dynamic behavior of the replicating Quine in a vessel of limited capacity $N$ is described by the *Catalytic Network Equation* (Stadler et al., 1993), a deterministic approximation expressed by Ordinary Differential Equations (ODEs) where $x_A$ is the number of A-molecules and where $x_B$ denotes the number of blueprints $B$

\[
\begin{align*}
\dot{x}_A &= \frac{r}{N} x_A x_B - \frac{x_A}{N} f(\vec{x}) \\
\dot{x}_B &= \frac{r}{N} x_A x_B - \frac{x_B}{N} f(\vec{x})
\end{align*}
\]

subject to the conservation relation $x_A + x_B = N$, where $N$ is the vessel capacity. The two molecules react, according to the law of mass action, with rate $r = x_A x_B$; each reaction event leads to an additional pair of molecules. The dilution flux applied to the vessel is equal to the net production rate $f(\vec{x}) = 2x_A x_B$, thus satisfying the conservation relation. The dilution flux is non-selective, this is, each species is diluted with a rate proportional to its relative concentration.

Molecular quantities in a limited vessel are often expressed in (relative) concentrations. The "concentration" of molecule $s \in S$, $\chi_s$, denotes the frequency of $s$ in the vessel multiset, hence
\( \chi_s = x_s / N \). Expressed in concentrations, the above equations become

\[
\begin{align*}
\dot{\chi}_A &= N \chi_A \chi_B - \chi_A \Phi(\vec{\chi}) \\
\dot{\chi}_B &= N \chi_A \chi_B - \chi_B \Phi(\vec{\chi})
\end{align*}
\]

where \( \Phi(\vec{\chi}) = 2N \chi_A \chi_B \).

The system exhibits three dynamic fixed points, one at \( \hat{\chi}_A = \hat{\chi}_B = 1/2 \) and two pathological cases at \( \hat{\chi}_A = 1, \hat{\chi}_B = 0 \), and \( \hat{\chi}_A = 0, \hat{\chi}_B = 1 \). The first fixed point is locally stable according to a standard perturbation analysis (Strogatz, 1994) and is characterized by both molecules – the blueprint and its active variant – being present with the same abundance.

Figure 4 demonstrates the stability of the replicating quine to perturbations: In a Fraglets vessel of capacity \( N = 1000 \) molecules both molecule types are present with 500 instances each. At time \( t = 0.1 \) s we forcefully removed 80% of the active molecules A from the vessel. The remaining Quines continue to produce replicas, which quickly repopulate the vessel. Once the vessel reaches saturation, there are more blueprints than active molecules, which causes the dilution flux to remove the first more frequently than the latter, until equilibrium is reached. At time \( t = 0.2 \) s we performed the same attack to the active molecules, from which the system recovers likewise.

![Figure 4: Replicating Quine under deletion attacks](image)

The stability property essentially means that the system returns to equilibrium condition: Even if we perturb the system by removing some instances of either species, the opponent forces of hyperbolic growth and non-selective dilution flux let the system autonomically find back to this fixed point. In other words, the system intrinsically maintains its own redundancy without an external controller!

5 A Generic Building Block for Self-Healing Software

The Quine studied so far just spends CPU cycles replicating itself. In this section, we demonstrate how our self-healing Quines can be enriched to perform an actual and useful computation. The
resulting data-processing Quine can be used as a software building blocks for various tasks, serving as a design pattern to engrath the self-healing property to an arbitrary piece of code.

5.1 The Data-Processing Quine

A small modification to the replicating Quine’s structure leads to a data-processing Quine, as is depicted in Figure 5. With the new structure, the set of A- and B-molecules does not directly replicate anymore. Instead, the active molecule A reacts with a data molecule (or “data packet”) D, computes some product and also generates an additional reward molecule R. The reward molecule reacts with and consumes a blueprint molecule B, which contains the necessary information to re-create the active molecule and its blueprint. As usual with replicating Quines, the reaction between R and B results in two instances of A and B.

Figure 5: Data-processing Quine: The active molecule A processes a data molecule D resulting in a reward R. The blueprint B contains all information necessary to generate two copies of A and B when reacting with R.

5.2 A Generic Template to Quinify Code

Every data-processing rule in Fraglets can be converted to a data-processing Quine, which is intrinsiically robust to the discussed faults. In the following we introduce a simple recipe to “quinify” code.

The data-processing Quine depicted in Figure 5 is the self-healing implementation of the following packet forwarding reaction:
The matchp-fraglet residing in node $v_1$ consumes local data packets destined to node $v_3$ and sends them to the next hop $v_2$. In fact, any persistent Fraglets rule $f$ like the one in (5) can be converted to a data-processing Quine by using the template

$$[\Psi_{\text{spawn}} f \Psi_{\text{consume}}(f) \Psi_{\text{replicate}} f \Psi_{\text{produce}}(f)]$$

(6a)

where generally the symbol strings $\Psi_{\text{spawn}}$ and $\Psi_{\text{replicate}}$ are defined as

$$\Psi_{\text{spawn}} := \text{fork} \ \text{nop} \ B$$

(6b)

$$\Psi_{\text{replicate}} := \text{split} \ \text{match} \ B \ \text{fork} \ \text{fork} \ \Psi_{\text{spawn}} \ *$$

(6c)

To generate the Quine replacement for $f = [\text{matchp} \ v_3 \ \text{send} \ v_2 \ v_3]$, we define the consumption and production part of of the data-processing Quine as

$$\Psi_{\text{consume}}(f) := \text{match} \ v_3$$

(6d)

$$\Psi_{\text{produce}}(f) := \text{send} \ v_2 \ v_3$$

(6e)

which results in the seed

$$[\text{fork} \ \text{nop} \ \text{match} \ v_3 \ \text{split} \ \text{match} \ B \ \text{fork} \ \text{fork} \ \text{nop} \ B \ * \ \text{send} \ v_2 \ v_3]$$

(7)

that bootstraps the data-processing Quine (compare Figure 5). Complex software consisting of many matchp-rules can be quinified by following this recipe. There are, however, some circumstances in which those Quines compete against each other for the limited memory rather than cooperate to accomplish a common task.

6 Competition Among Independent Quines

By applying an excessive dilution flux, we created a competitive environment in which software components have to rewrite themselves continuously. A single Quine does so and this enables it to survive. But software rarely consists of a single Quine. In this section we study whether multiple Quines are able to coexist in the same vessel under the selective pressure of the dilution flux. As expected, they naturally compete for the limited resource, leading to the extinction of all but one. However, when putting the Quines into the different vessels of a network, they may coexist in separated islands.
6.1 Competition in a Well-Stirred Vessel — The Winner Takes All

As we showed in the previous chapter, the aggressive growth of the Quine is instrumental for letting it self-heal. However, when placing two different instances of simple replicating Quines into a common reaction vessel, only one will survive while the other will be literally squeezed out. This is due to the finding that independently and hyperbolically growing populations with finite resources lead to the survival of the common (Szathmáry, 1991): The first Quine that reaches a sufficiently high concentration will dominate the others and lead to their extinction even if their replication rate is higher.

Figure 6 illustrates the struggle between two replicating Quines in the same vessel for different initial concentrations but a constant total vessel capacity of \( N = 20 \) molecules. We observe that the robustness of the Quines heavily depend on their initial concentration – the (initial) allocation of memory slots for Quine 2 grows linearly from left to right – and that the non-selective dilution flux does not guarantee fairness among the Quines per se as one of the Quines is squeezed out by the other within a fraction of a second.

![Figure 6: Coexistence time and survival probability of two competing Quines](image)

Figure 6: Coexistence time and survival probability of two competing Quines: The two replicating Quines do not coexist; the one with a higher initial concentration prevails most likely. Top: Mean coexistence time in a vessel of capacity \( N = 20 \) molecules, plotted with respect to the relative initial concentration of the second Quine. Bottom: Survival probability of either Quine in the same vessel.

6.2 Competition in the Network — Separated Islands

In a distributed setting, the outcome is different: different Quines may permanently coexist and form separated islands.

We study a modified version of the replicating Quine that sends one replica to a neighbor node: Each network node runs a separate reaction vessel and schedules the reaction according to a law of mass action scheduler. An active molecule \( A_i \) of each node \( i \) reacts with the local blueprint \( B_i \), and their product splits into two parts: The first fragment regenerates the Quine by producing the local molecules \( A_j \) and \( B_j \). A second Quine seed is sent to an arbitrary neighbor node \( j \), where it unfolds into the molecules \( A_j \) and \( B_j \). In Fraglets, this is realized by encoding
the two species as

\[ A_i = \text{[match B fork nop anycast fork nop match B]} \] \hspace{1cm} (8a)
\[ B_i = \text{[B fork nop anycast fork nop match B]} \] \hspace{1cm} (8b)

We studied this distributed Quine in different network topologies and found out that the network fosters phase islands, in which different Quines may coexist in spatially distant locations.

Figure 7 shows the results of a typical Fraglets simulation run in a network of 100 vessels arranged in a two-dimensional grid. Each vessel has a capacity of \( N = 100 \) molecules and is initially populated with an equal quantity of the two Quines. The two figures show a series of spatial snapshots of the network over time. Each pixel represents the vessel at the corresponding coordinates: white and black colors indicate that either Quine 1 or Quine 2 dominates whereas a coexistence of both Quines is represented by gray pixels.

For the first example, we used a toroid network topology, meaning that vessels at the border of the grid are connected to the vessel at the opposite border. Figure 7(a) depicts a pattern we frequently observed in this topology: stationary horizontal or vertical stripes of either Quine type. A vessel inside such a stripe is completely surrounded by vessels hosting the same Quine type. At the border of the stripe, the two Quines coexist in the same vessel, because the inflow of either type from opposite directions is approximately equal.

If we slightly reconfigure the network topology by disabling the wrap-around connections at the border we observe additional patterns, such as the one depicted in Figure 7(b): One Quine retracts and is able resist the dominance of the other Quine in one of the corners of the network.

It is already well known that bistable systems become stable in a spatial environment (Kapral & Oppo, 1986; Károlyi, Péntek, Scheuring, Tel, & Toroczkai, 2000) or are able to form more complex organizations (Dittrich & Banzhaf, 1997). Additionally, there are results from evolutionary graph theory (Lieberman, Hauert, & Nowak, 2005) indicating that the topology of the network graph has a strong influence on the selective pressure. A more detailed analysis of the spatial behavior of Quines is left for future work.

Note that in these experiments, we initialized all nodes with the same amount of either Quine, and therefore gave both Quines the same chance to prevail. Once the network only hosts one Quine the probability that the network is invaded by another similar Quine is very low. In biology, this is called the fixation probability, which denotes the probability that a population of wild-type instances is replaced by a single instance of a mutant (Wright, 1931; Moran, 1958; Nowak, 2006). Hence, once installed, it is very hard to get rid of a Quine or to replace it with a newer version. In Section 8 we demonstrate how two versions cooperate such that a distributed software update is possible.

6.3 Competing Data-Processing Quines

We return to the analysis of competition in a well-stirred vessel. The struggle for existence between different data-processing Quines is behaviorally similar to the replicating Quines although they need data packets to replicate.

For moderate packet injection rates we distinguish two different scenarios depending on whether or not two Quines process the same packet (see Figure 8): (a) When consuming the
same molecular species (for example by matching the same header tag in Fraglets) the two Quines actually fight for the same “food” molecule D, which is required for the Quines to replicate. Due to the law of mass action the Quine exhibiting a larger multiplicity will react more often with the food, yielding even more offspring. Consequently, the qualitative outcome is the same as for the simple replicating quine: survival of the common. (b) Even if the two Quines are “digesting” distinct food molecules, D1 and D2, respectively, they still live in the same habitat and fight for the same memory resource. The Quines only coexist if their food molecules are injected with a rate in the same order of magnitude, which causes their replication rate to be similar. In fact, the steady-state concentration of the Quines is proportional to the corresponding food injection rates. Otherwise, if the injection rates diverge too much, the concentration of the slower Quine will very likely drop to zero and the Quine becomes extinct.

This competing behavior is problematic because we would like to compose multiple self-healing Quines, each working on some part of a problem to solve. As we show in the next section, this is in fact possible if the Quines cooperate.

7 Cooperative Linkage of Quines

In this section we introduce four design patterns to couple data-processing Quines. Cooperating Quines must mutually control their replication rate in order to achieve fairness with respect to their expected survival time, which we want to be independent from their packet processing rate. In the following we analyze various coupling methods for data-processing Quines.
7.1 String of Quines

Multiple data-processing Quines are indirectly linked by sequentially processing the same data stream.

Frequently, a computation requires multiple data-processing operations in sequence. Figure 9 shows such a scenario for $n$ operations, each using our data-processing Quine structure. In such a reaction network topology, the common packet stream generates a symbiotic relationship among the Quines: The second Quine is only able to replicate if the first Quine generated a product molecule and replicated in turn. The third Quine only replicates after the second replicated before. Thus, the growth rate of all Quines is approximately equal to the data injection rate.

7.2 Hypercycle

If Quines do not process the same data stream sequentially, we need another method of symbiotically coupling the Quines. Eigen and Schuster (1979) proposed a cyclic linkage of reactions as an explanation of self-organization of prebiotic systems in which RNA strands and enzymes cooperate. We translate this idea to the world of Quines in Fraglets: As shown in Figure 10, a cycle of Quines consists of several data-processing Quines, which do not generate their own replication reward $R_i$. Instead, each Quine cyclically generates the reward for another Quine.

The advantage of such a cyclic dependence is that none of the Quines is able to replicate much faster than the others, which leads to their sustained coexistence in the reaction vessel. The disadvantage comes from the fact that an active molecule $A_i$ is consumed when a data packet is processed and is not immediately available for the next data packet. Thus, on the long run, Quine $i$ cannot process a data stream faster than twice the data processing rate of the...
predecessor Quine. This is, the hypercyclic Quines impose strong restrictions on the range of data streams they are able to process as the most active element is doomed to starve.

7.3 Data-Rate Independent Replication Feed

The main disadvantage of the hypercylic Quine, starvation of the most active molecules, arises from the fact that these molecules are produced by one specific other Quine in the cycle. This dependency can be broken by separating regeneration from replication: When processing a data packet, the Quine shall immediately regenerate the consumed active molecule. This maintains a constant number of active molecules, provided that there is no dilution flux and no (mutation or execution) error. To cope with the relatively rare error events we have to provide a separate stream of replication rewards for the fraglet; these replication rewards increase the number of active molecules and blueprints. Because we want all coupled Quines to survive, we have to guarantee that they all receive their replication rewards with the same rate, i.e., they are fairly fed.

Figure 11: Separation of regeneration and replication: The two data-processing Quines only regenerate themselves; a replication feed is provided by a separate supporting Quine on the right.

Figure 11 shows such a replication feed pattern. Two Quines process independent data streams and regenerate themselves individually by generating a local regeneration reward
molecule $R_g^i$ for each data molecule processed. A separate support Quine generates replication rewards $R_r^i$ and distributes them uniformly to the involved Quines. The support Quine is self-replicating, paced by an externally provided stream of trigger molecules. Ideally, its “feeding rate” would be adjusted to the expected error rate. If the error rate cannot be estimated, the data processing Quines could be reprogrammed in order to automatically produce this trigger, although the latter method will probably result in a feeding rate that is higher than needed.

### 7.4 Compartmentation

Another way to switch-off competition among two individuals is to put them in different habitats. If two Quines are located in different vessels, each with separate memory resources, their growth do not affect each other. This is a simple yet very effective method to separate Quines operating on different packet streams.

Compartmentation is also helpful to increase the survival time of data-processing Quines in general. We previously found out that the robustness of the data-processing Quine decreases for increasing packet rates because packets eventually displace all other molecules (Meyer & Tschudin, 2010). We suggested to increase the vessel capacity, which shifted the critical packet injection rate towards higher values.

![Figure 12: Additional replication feed from a separate vessel](image)

Instead of adding more resources in order to increase the robustness of the data-processing Quine, we may use the pattern depicted in Figure 12. A supporting Quine, similar to the one seen previously, now lives in a separate reaction vessel. The supporting Quine is fed by an external trigger, which initiates the replication of the supporting Quine and additionally generates a seed that is required to bootstrap the data-processing Quine in the outer vessel. The latter Quine lives in a harsh environment subject to an uncontrollable packet stream, which may cause the death of the Quine. However, the isolated supporting Quine is not affected by the packet stream and eventually replenishes the $A_1$- and $B_1$-molecules of the data-processing Quine.

If we study the data-processing Quine in the outer vessel of Figure 12 we recognize that all species except the reward $R_1$ are injected from externally. The reward molecule itself is produced in a reaction among $A_1$- and $D_1$-molecules. It is no surprise that the outer reaction vessel is always able to survive; it is able to recover from the loss of every and all species. Consequently,
the survival time of the overall system is equal to the survival time of the supporting Quine in the inner vessel. Since the supporting Quine is not affected by the packet stream, its survival time only depends on the trigger molecule injection rate \( r_{\text{in},s} \) and the vessel capacity of the inner vessel \( N_s \). A detailed analysis of the survival time of a Quine is given in Meyer and Tschudin (2010).

8 Application Case: Self-Healing Distributed Software Updates

The distributed Quine, introduced in Section 6.2, can be used to deploy software in a network: One Quine instance put in single vessel quickly spreads over the whole network. But we also mentioned before that a distributed Quine is very robust to the invasion of a competitor. Once installed, distributed software built with this pattern is hardly ever exchangeable. Often a network operator – or in the future, the system itself – has to install software updates. For self-healing software we also need a method to replace all or certain building blocks. In this section we provide a method to construct distributed, updatable software, which is still inherently self-healing.

Thereby we revisit three ideas mentioned so far and combine them to obtain a robust software deployment pattern: First, we assume that all networking software consists of data-processing Quines and is therefore intrinsically robust to certain code-level perturbations. Second, we combine this pattern with the distributed Quine, introduced in Section 6.2, which increases the overall robustness of the software by distributing replicas over the network. Third, all Quines are tagged with a version number; Quines with a higher number shall replace Quines with a lower number. (If the networking software comprises of several cooperative Quines, each of them has its own versioning and can be updated separately.) If we manage to unify these three elements in a single Quine structure the resulting distributed software autonomically updates itself: The distributed replication mechanism propagates old and new software versions while locally interacting old and new Quines destroy the first and proliferates the latter.

In Section 8.1, we present our self-healing Quine variant that performs some useful function, autonomically disseminates itself over the network, and updates itself with a coexisting newer version. Then, in Section 8.2 we demonstrate the feasibility of this approach with the help of Fraglets simulation results.

8.1 A Versioned Quine

Figure 13 depicts the reaction network of the versioned Quine. We assume that multiple versions \( i \) of the Quine may coexist in the same vessel for a short time. They process the same data molecule \( D \) but, because of their different blueprints \( B_{vi} \), their active variants \( A_{vi} \) produce different results.

At first glance this pattern looks like the competing parallel Quines discussed in Section 6 (see Figure 8(a) on page 15). We mentioned that parallel Quines competing for the same “food” molecule inevitably lead to the extinction of all but one. This is in fact what we are aiming at, but we want to control which of the Quines is able to survive, namely the most recent one. Unlike the parallel Quines studied before, the molecules of the versioned Quine interact. More precisely, the reward \( R_{vj} \) only replicates newer blueprints \( B_{vj} \) and their active variants \( A_{vj} (j \geq i) \). Otherwise, if a reward molecule reacts with an older blueprint, the two molecules annihilate.
Figure 13: Versioned Quine: Multiple versions of the same Quine compete for the same data molecule D. Unlike in Figure 8(a) (parallel Quines), their rewards $R_{v1}$ and blueprints $B_{v1}$ interact. Rewards eliminate old blueprints but replicate its own or newer blueprints.

each other. This mechanism provides newer software a selective advantage over the older versions.

8.2 Fraglets Simulation Results

We implemented the versioned Quine in Fraglets and performed extensive simulations to study its behavior. One particular topology we studied in more detail is the toroidal network of 100 nodes: a regular, fully connected network where each node has four neighbors and a capacity of $N = 100$ fraglets. Starting with empty nodes we injected a single Quine of version 1 into one of the nodes. As depicted in Figure 14 the Quine quickly distributes itself over the network. Each image in the figure shows the mixture of Quine versions in the two-dimensional network topology. This is, each pixel represents the node at the corresponding coordinates. We chose a color scheme based on the normalized number of blueprints $\alpha$, defined as

$$\alpha = \frac{N_{B_{v2}} - N_{B_{v1}}}{N_{B_{v1}} + N_{B_{v2}}} \quad (-1 \leq \alpha \leq 1)$$  \hspace{1cm} (9)

If a node only contains version 1 Quines, the value of $\alpha$ is $-1$ (white), if it is completely filled with version 2 Quines, $\alpha$ is $+1$ (black), and if the two Quines are present with equal concentration, $\alpha$ has a value of 0 (gray).

Once all vessels were populated by Quine version 1 we flushed an arbitrary vessel with version 2 instances by injecting $N$ seeds with the new code. Figure 15 shows how the new Quine populates the network: In a first phase ($t < 1$ s), the new Quine spreads to the neighborhood of the source node. While annihilating version 1 blueprints, the new Quine is diminished by version 1 Quines that surge in from the neighbor nodes. In a second phase ($t < 7$ s), the new Quines spread over the whole network but still has a very small concentration compared to the old version. Suddenly, around $t \approx 7$ s, a region distant from the original source node exhibits a substantial concentration of the new Quine. From this location the new Quine quickly takes over the network and eventually prevails.
inject one \( v_1 \)-seed

\[ \alpha = -1 \]
\[ \alpha = 0 \]
\[ \alpha = +1 \]

more \( v_2 \)

Figure 14: Initial spreading of the versioned Quine: Series of snapshots over time in a toroidal topology of 100 nodes for a Fraglets simulation of the versioned Quine. The Quine quickly populates all vessels in the network if no software is installed yet.

flush with \( v_2 \)-seeds

\[ \alpha = -1 \]
\[ \alpha = 0 \]
\[ \alpha = +1 \]

more \( v_2 \)

more \( v_1 \)

Figure 15: Infection with version 2 Quines and their pervasion: Series of snapshots over time in a toroidal topology of 100 nodes for a Fraglets simulation of the versioned Quine. A newer version is spread but attacked at the same time by the inflow of older versions from neighbor vessels. However, eventually the version 2 Quine prevails.

Finally, as a cross-check, we flushed the same source vessel with version 1 instances once the network only hosted version 2 Quines. As shown in Figure 16, older Quines are not able to displace more recent software.

8.2.1 The Update Resistance of Large Networks

The versioned Quine behaves like a distributed chemical switch (Ramakrishnan & Bhalla, 2008). If a newer version of the same Quine is present with a certain threshold concentration, the whole network flips to the new software.

Our experiments revealed that this threshold depends on the size of the network. The more old-versioned Quines put pressure against the injected update, the less likely it is that the software update is successful. We showed before that the update succeeds in a network of 100 nodes when completely replacing the older versions by new version instances in a single node. However, already in a network of 400 nodes, the same strategy always leads to the extinction of
The new software. The new Quines diffuse too fast over the network and are not able to build-up the critical concentration needed for triggering the transition.

One solution that worked for all (tested) network sizes was to continuously inject the new Quine for a certain time period with a rate that is approximately equal to the vessel size (see Figure 17). By continuously seeding the new Quine we help it getting over the critical threshold and forming a spatially growing topological island of new versions. At the center of the island, all neighbors are populated with new Quines: the island maintains itself. At the border, the inflow of old Quine versions is limited due the neighborhood to nodes in which the new Quine dominates. Consequently, the island expands and eventually covers the whole network.

9 Conclusions

In this report we further elaborated our chemical execution model in which programs organize their own redundancy in order to achieve a high code-level robustness: Quines, i.e. self-replicating artificial code molecules, are able to procreate and survive in an environment with
limited memory resources. We highlighted the emergence of competition and showed several strategies how to link the Quines, such that they cooperate.

Unlike other bio-inspired approaches that extract the essence of a natural mechanism and then try to import it into existing networking machinery, we make a step forward by putting chemical laws in the core of software execution engines: It seems that such an approach simplifies the transfer of desirable robustness properties from nature to manmade systems. Software robustness must be understood as a code-rewriting task with homeostatic properties where the software continuously and actively has to maintain its healthy state. We believe that this is a necessary requirement for the future’s truly adaptive and evolving software. However, we conjecture that the adoption of nature-inspired mechanisms to achieve self-* properties always comes along with emergent opponent forces that need to be tamed.

Acknowledgments

This work has been supported by the Swiss National Science Foundation through SNF Project Self-Healing Protocols (2000201-109563). We would also like to thank Lidia Yamamoto for her continuous contribution to our research.
References


