Programming by Equilibria

Christian Tschudin, Thomas Meyer

Computer Science Department, University of Basel
Bernoullistrasse 16, CH–4056 Basel, Switzerland
{christian.tschudin,thomas.meyer}@unibas.ch

Abstract. We present a reactive computing paradigm that seeks to solve problems by bringing the programmed system into an equilibrium state: The production of a certain (numeric) result, or of a certain side effect (routing), emerges from the system's tendency to strive for an equilibrium. We have identified important equilibria for properties like adaptivity, as well as self-healing where a program controls its own program code. By linking our execution model with well known laws from chemistry, we can predict a program's dynamic behavior and in some cases even provide elegant proofs of convergence. In this paper we recapitulate how one can compute results out of the random execution of instructions. We then introduce an artificial chemistry as our reactive programming environment, in which some networking protocols can be expressed in a natural way. We present a gossip-style protocol for the cooperative computation of an average value which is amenable to a formal stability analysis, as well as a load balancing protocol implementation that is self-healing.

Keywords: reactive programming, artificial chemistry, computer network protocols, provable program dynamics, self-healing.

1 An Introductory Example

Consider a checker board with two types of tokens, white and black. In each round, apply a simple rule: Randomly select two places and compare the color of the tokens found there. If they are identical, do nothing. If the colors are different, create a copy of each token and let them drop at random places (replacing any previously present token color). Then start the next round. We call this variation of Eigen's game [1] the "mate-and-spread game".

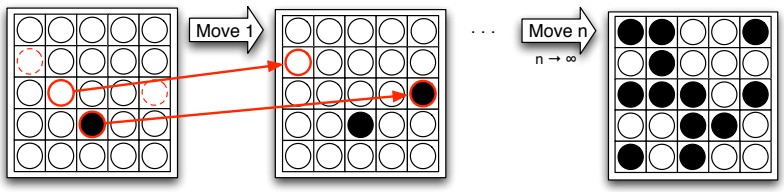

Fig. 1. The board is initialized with only one black token and 24 white tokens. After some rounds, the number of white and black tokens mutually approach each other.

If each color is presented at least once in a start configuration, the resulting system will always be the same: No matter of the initial distribution of the tokens, this game strives for an equilibrium where black and white tokens are present with the same abundance as shown in Fig. 2.

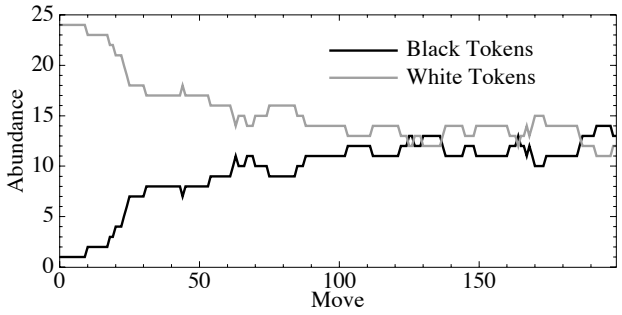

Fig. 2. In the mate-and-spread game, the amount of each token type eventually fluctuates around equality.

The microscopic process of a move in this game is random (random selection and random replacement) and we cannot predict the outcome of the next round. However, at the macroscopic level we observe the emergence of an equilibrium.

1.1 An Example of Programs in Equilibrium

We can produce the same system behavior if we turn the passive tokens into active *programs*. Instead of a central player picking tokens and replacing them if needed, tokens try to bind to (mate with) each other and, if necessary, replace themselves as well as other tokens.

Consider the two "programs" `[match x fork fork fork nop match x]` and `[x fork fork fork nop match x]` that represent our tokens from before. Their respective prefixes (see Section 2 for full details) makes them to bind with each other, producing the following intermediate program:

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

The new program's prefix, `fork`, requests replacement by two copies of itself, with the second and third symbol being the new header symbols. Surprisingly, both resulting programs will be identical and read as follows:

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

The next step, for each of these programs, is again the copy-yourself operation, leading to two different programs, namely `[nop x fork fork fork nop match x]` and `[match x fork fork fork nop match x]`. The `nop` prefix is simply executed (doing nothing), and we end up with the two programs we started with, except that now we have them twice.

These programs are "Quines"[1], as they regenerate, and even duplicate themselves. This system would grow unboundedly, wherefore we add the rule that new programs (created out of `fork` or `match`) replace other programs in the system randomly. Such a system will behave like the mate-and-spread game from above, always approaching an equilibrium where both types of programs are present in the same quantities. That is: even if we destroy all copies of one program type except one, the system will recover and "heal itself".

As simply as the logic of these prefix programs looks, there are still some elements to examine in more detail. Does, for a given set of initial program types, an equilibrium exist? Such proofs, especially as they relate to the dynamics of program execution, are quite hard to achieve in ordinary sequential programming. By studying our prefix programs above as a chemical reaction network, however, we can use well-known perturbation analysis tools to predict the macroscopic behavior, **iff** we carefully craft the scheduling of program execution.

1.2 Structure of this Paper

In Sect. 2, we expand a little more on the prefix programming language used in our approach and demonstrate a simple computation task for an artificial chemistry. Section 3 introduces a program scheduling that observes the chemical "law of mass action". Section 4 presents a load balancing protocol for which we prove the existence of the desired equilibrium and which, moreover, is a self-healing protocol implementation.

2 Prefix Programs (Fraglets)

We call tiny programs that bind with each other and rewrite themselves "fraglets", which stands for a small fragment of a computation. Fraglets react with other fraglets akin to molecules in chemistry. In this section, we provide a condensed description of the Fraglets language [2, 3], an artificial chemistry [4], whose corresponding chemical machine is executable and which serves as a simple platform to run "chemical programs".

Each fraglet $s \in \mathcal{S}$, is a string of symbols over a finite alphabet Σ. The first symbol of the string defines the string rewriting operation applied to this molecule by the virtual chemical machine: The prefix can be thought of an assembler instruction. The following list shows some of these instructions and their actions:

[1] after the philosopher and logician Willard van Orman Quine (1908–2000) who studied indirect self-reference

$$
\begin{aligned}
\texttt{[match } \alpha\ \Phi\texttt{]} + \texttt{[}\alpha\ \Omega\texttt{]} &\longrightarrow \texttt{[}\Phi\ \Omega\texttt{]} \\
\texttt{[matchp } \alpha\ \Phi\texttt{]} + \texttt{[}\alpha\ \Omega\texttt{]} &\longrightarrow \texttt{[matchp } \alpha\ \Phi\texttt{]} + \texttt{[}\Phi\ \Omega\texttt{]} \\
\texttt{[fork } \alpha\ \beta\ \Omega\texttt{]} &\longrightarrow \texttt{[}\alpha\ \Omega\texttt{]} + \texttt{[}\beta\ \Omega\texttt{]} \\
\texttt{[nop } \Omega\texttt{]} &\longrightarrow \texttt{[}\Omega\texttt{]} \\
\texttt{[send } k\ \Omega\texttt{]}_i &\longrightarrow \texttt{[}\Omega\texttt{]}_k \quad \text{if } i \text{ and } k \text{ are neighbor nodes}
\end{aligned}
$$

$\alpha, \beta \in \Sigma$ are arbitrary symbols, $\Phi, \Omega \in \Sigma^*$ are symbol strings and $i, k \in V$ are network nodes, or metaphorically, reaction vessels. Molecules starting with `match` or any non-instruction identifier are in their *normal form*. The `match` instruction can be used to join two molecules inside the same vessel 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` n_2 `pkt]` and `[pkt data]` in node n_1 imply the following reaction:

$$
\begin{aligned}
\texttt{[match pkt send}\, n_2\, \texttt{pkt]}_{n_1} + \texttt{[pkt data]}_{n_1} & \\
\longrightarrow \texttt{[send}\, n_2\, \texttt{pkt data]}_{n_1} & \\
\text{(transmission to neighbor } n_2\text{)} \quad \longrightarrow \texttt{[pkt data]}_{n_2} &
\end{aligned}
$$

Such a chemical language allows us to "program" a reaction graph: Computation is realized by discrete string rewriting steps inside a node, but also across a network of nodes, spanning a global reaction network.

2.1 Chemically Computing an Average Value

In the following we present a chemical implementation of a gossip-style protocol in Fraglets that makes use of a chemical equilibrium in order to compute the average of values stored in distributed nodes (see [5] for other gossip protocols).

In each node of a network, and for each of its neighbors k we inject one instance of the following "shuttle service" fraglet: `[matchp X send` k `X]`. The input value to the computation is represented by the initial number of X molecules. Thus, in each node, we place as many X-molecules as the x value reads. The link-specific shuttle service fraglets[2] compete for the local X molecules in a random fashion and, when reacting, transfer one of them to the corresponding neighbor node. These fraglets build a virtual reaction network that spans over all network nodes.

In order to illustrate the principle of our protocol, we carried out an OMNeT++ [7] simulation in the network depicted in Fig. 3. We initialized node 1 with 1000 molecules of type X; all other nodes contain no X molecules at the beginning of the simulation. Figure 4 shows how the value of each node asymptotically converges to the expected average of $x_i(t \to \infty) = 250$ (see Sect. 3.2, and [8] for a formal convergence proof).

[2] For a variant of the protocol where the nodes do not require information about their neighbors, see [6]

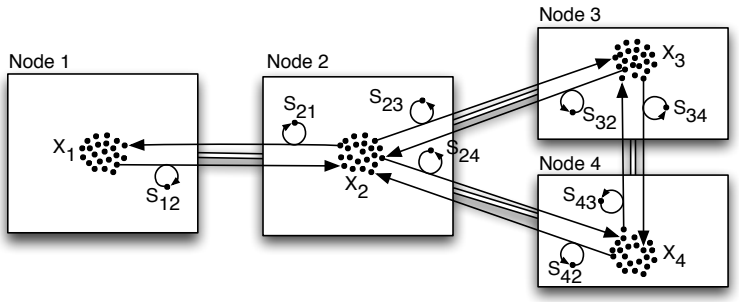

Fig. 3. The chemical protocol calculates the average concentration of X molecules in a network of virtual reaction vessels.

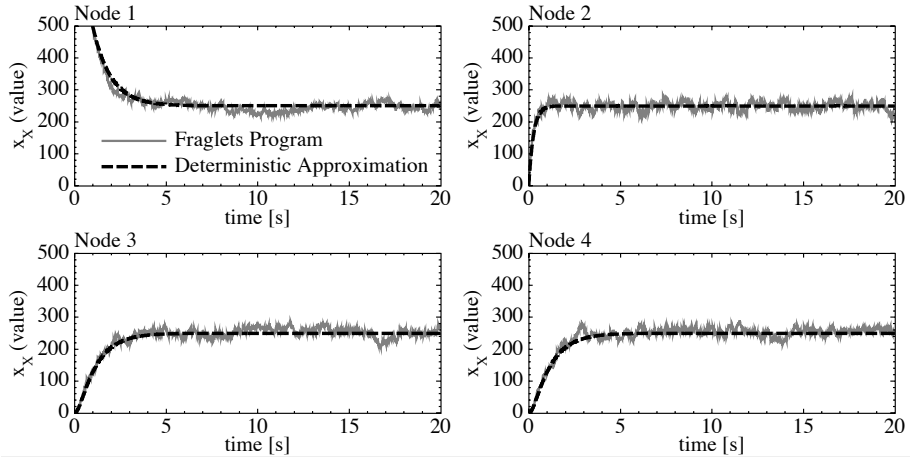

Fig. 4. Simulation of the chemical averaging protocol (Fraglets program run), and analytic prediction using a deterministic ODE model.

3 "Chemical" Scheduling and System Equilibrium

In the previous, distributed computation example, the system would not strive for an equilibrium if all network nodes would send X-molecules as fast as possible. An appropriate scheduler has not only to decide which reaction is executed next, but also *when* the reaction occurs. In this section we present a scheduling algorithm for Fraglets that helps us designing equilibrium solutions.

One scheduling algorithm that was proposed by Gibson and Bruck [9] for artificial chemistries, is based on laws discovered in statistical mechanics [10]: Molecules in a reaction vessel undergo Brownian motion, which leads to collision events. The more molecules there are in a given volume, the more frequently they collide. Consequently, in a reaction vessel of constant volume, the reaction rate is proportional to the product of the reactant's concentrations. This relation is known as the *law of mass action* [11].

The algorithm of Gibson and Bruck, which we adopted for our scheduler, calculates the next occurrence time for each reaction r individually. First it counts the combinatorial number of potential collision partners, i.e. the product of the reaction's reactant (input molecule) concentrations:

$$a_r = \prod_{i=1}^{n} x_i^{\alpha_{ir}} \tag{1}$$

where n is the number of molecule types, x_i is the concentration of molecule type i, and α_{ir} denotes the number of molecules of type i that are consumed by reaction r.

The Gibson-Bruck algorithm then draws an exponentially distributed random variable to determine the reaction interval of r based on the number of possible collision partners:

$$\tau_r \sim \mathrm{Exp}\left(\frac{1}{a_r}\right) \tag{2}$$

After reaction r has been executed at time t_{now}, its next occurrence time is computed as

$$t_{r,\mathrm{next}} = t_{\mathrm{now}} + \tau_r \tag{3}$$

In our implementation the so calculated occurrence times of all reactions are sorted into a priority queue. In a simple loop, the scheduler continuously extracts the first element from the queue, executes the corresponding reaction and computes the new reaction time according to (3). Between two reactions, the reaction vessel sleeps for a well-defined, but inherently stochastic, idle time.

3.1 Deterministic Approximation of the System Dynamics

Despite the randomness of the reactions on the microscopic level, and *because* of the forced idle time of the CPU, we can make predictions about the dynamic behavior of such an artificial chemistry. We approximate the reaction rate by only considering the mean value of the reaction interval $\langle \tau_j \rangle$:

$$v_r = \prod_{i=1}^{n} x_i^{\alpha_{ir}} \tag{4}$$

This rate reflects the macroscopic *law of mass action*. We translate the discrete set of reactions $\mathcal{R}$ into a continuous ODE (ordinary differential equation) model:

$$\dot{\boldsymbol{x}}(t) = \mathbf{N}\boldsymbol{v}(t) \tag{5}$$

where $\dot{\boldsymbol{x}}(t) = \left(\dot{x}_1(t)\ \dot{x}_2(t) \cdots\ \dot{x}_n(t)\right)^T$ is the vector of the time derivatives of all molecule concentrations, $\mathbf{N}$ is the stoichiometric matrix ($\mathbf{N} = [\gamma_{ij}]$ where γ_{ij} denotes the net number of molecules of type s_i that are generated by reaction

r_j), and $\boldsymbol{v}(t)$ is the vector of reaction rates $\boldsymbol{v}(t) = \big(v_1(t)\ v_2(t) \cdots\ v_m(t)\big)^T$ with v_r according to (4).

Based on such a traditional description of chemical kinetics, we are able to analyze the system's behavior and analyze properties like the development of concentrations in time, finding equilibria and analyzing their stability.

3.2 Perturbation Analysis

From the ODEs of a reaction system, there is a small step towards finding its equilibria (or fixpoints). If the chemical reaction system strives for an equilibrium, the number of molecules in equilibrium, $\boldsymbol{x}^*$, do not change anymore. Thus, in (5) we set $\dot{\boldsymbol{x}} \equiv 0$. For the averaging example in Sect. 2.1, we can show that there is a single fixpoint

$$x_i^* = \frac{\sum_{k=1}^{N} x_k^*}{N} = \langle x^* \rangle \tag{6}$$

where the value x_i in each node equals the average value (N: number of network nodes).

To prove that the fixpoint is stable when disturbed by a small perturbation, we linearize the system around the fixpoint by calculating the Jacobian matrix:

$$J = \left[\frac{\partial \dot{x}_i}{\partial x_j}\right] \tag{7}$$

The fixpoint is stable if the real part of all eigenvalues of the Jacobian, evaluated at the fixpoint, are smaller than zero. See [8] for the complete stability analysis of the averaging example.

4 A Self-Healing Protocol

In Sect. 1 we introduced a simple self-duplicating Quine in Fraglets. Such a Quine can easily be converted into one that processes some data stream while replicating itself (see [12] for more details). We now make use of such Quines as building blocks in order to assemble a self-healing protocol that balances a packet stream over two different network paths such that packet loss is minimized.

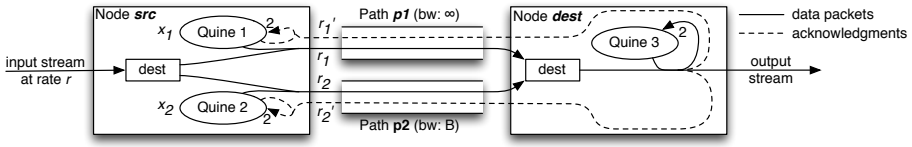

Fig. 5. In the self-healing load balancing protocol, packet streams control the replication of code.

As depicted in Fig. 5, we inject packets (=fraglets) at rate r into the source node where two quines, one for each path, compete for them and send them over

the corresponding paths. Instead of replicating as fast as possible, the quines depend on and react with acknowledgment packets. These acknowledgments are sent back over the reverse path by the third quine in the destination node, which also delivers the original packets to the data sink.

4.1 Formal Convergence Proof

This simple scheme leads to a perfect packet balance among the two paths, matching the available bandwidths. When the source node's reaction vessel is saturated, its molecules either belong to quine 1 or 2, as other molecules have been squeezed out. Let's denote the relative concentrations by x_1 and x_2, respectively, satisfying $x_1 + x_2 = 1$. Since replication is triggered by received acknowledgments, these concentrations are $x_1 = r'_1/(r'_1 + r'_2)$ and $x_2 = r'_2/(r'_1 + r'_2)$ where r'_n is the rate of acknowledgments received over path pn.

Let's assume that the bandwidth of *p1* is infinite whereas *p2* drops packets exceeding a rate of B packets/s. We examine the overload situation where the total rate $r > 2B$. Consequently, the rate of acknowledgments is $r'_1 = r_1$ and $r'_2 = \min(r_2, B)$. Due to the law of mass action, the fraction of packets sent over *p1* is proportional to the concentration of quine 1: $r_1 = x_1 r = r_1 r/(r_1 + \min(r_2, B))$. Hence

$$r_1 = r - B \quad \text{and} \quad r_2 = r - r_1 = B \tag{8}$$

Quine 2 reduced its concentration so as to only forward packets up to the bandwidth limitation of path *p2*, as was to be proved.

4.2 Surviving Code Attacks

Like in the computation example based on the gossip-style protocol, our load balancing protocol reaches a (chemical) equilibrium. Deviations from the equilibrium, for example by lost molecules on a network path, are compensated by increasing the population of quines that forward packets over the opposite path. Moreover, the code itself is organized in circuits of self-replicating molecules: If we forcefully destroy some code, the system will eventually regenerate it and, after some transition time, autonomously finds back to the equilibrium state. In fact, packet loss as well as *code loss* are treated by the same mechanism and in the same way.

Figure 6 depicts the protocol's response to such a code deletion attack[3], simulated in OMNeT++ for $r = 200\,\text{pkts/s}$ and $B = 40\,\text{pkts/s}$ and the network topology shown in Fig. 5. At $t = 50\,\text{s}$ we remove 80 % of all molecules from the source vessel: The code regenerates itself within 10 s while the traffic distribution r_1/r_2 remains unchanged.

[3] Code mutations can also be caught by mapping them onto code deletion events using a simple instruction encoding guarded by a parity-bit.

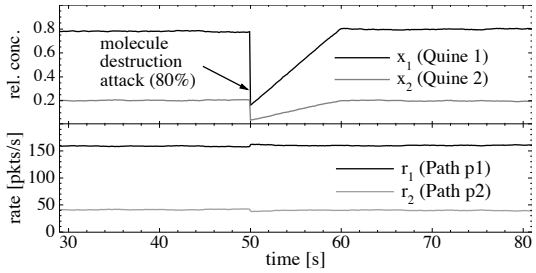

Fig. 6. Simulation of self-healing load balancing quines: Relative concentration of forwarding code in node *src* during a deletion attack, and the corresponding packet forwarding rate (unaffected) of the quines.

5 Conclusions

Representing a solution to a computation as an equilibrium of a dynamic system, lets the system "automatically" strive for this solution, starting from an unbalanced initial configuration (=input value) and tolerating perturbations during the execution.

We have demonstrated in this paper such an approach by showing a prefix programming language that permits to define artificial chemical reaction networks. Imposing a special scheduling of discrete instructions that is compatible with the "law of mass action", we obtain systems that can be described by continuous functions and whose dynamics is amenable for a formal perturbation analysis.

Finally, by abandoning the strict distinction between code and data, one can create reaction networks which regulate their own "code replication" as for example produced by Quine programs. This permits to write programs that provably strive for code equilibrium, which is another way of saying that a program has become self-healing.

Acknowledgments

This work has been partially supported by the Swiss National Science Foundation and the European Union, through SNP Project Self-Healing Protocols and FET Project BIONETS, respectively.

References

1. Eigen, Manfred und Winkler, R.: Das Spiel. Piper (1975)
2. Fraglets home page `http://www.fraglets.net`.
3. Tschudin, C.: Fraglets - a metabolistic execution model for communication protocols. In: Proc. 2nd Annual Symposium on Autonomous Intelligent Networks and Systems (AINS). (2003)
4. Dittrich, P., Ziegler, J., Banzhaf, W.: Artificial chemistries - a review. Artificial Life **7**(3) (2001) 225–275

5. Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: Proc. 44th Annual IEEE Symposium on Foundations of Computer Science. (2003) 482–491
6. Meyer, T., Yamamoto, L., Tschudin, C.: An artificial chemistry for networking. Volume 5151 of Lecture Notes in Computer Science. Springer, Berlin / Heidelberg (2008) 45–57
7. Omnet++ homepage `http://www.omnetpp.org`.
8. Meyer, T., Tschudin, C.: Chemical networking protocols. In: Proc. 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII). (2009)
9. Gibson, M.A., Bruck, J.: Efficient exact stochastic simulation of chemical systems with many species and many channels. Journal of Physical Chemistry A **104**(9) (2000) 1876–1889
10. Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. Journal of Physical Chemistry **81**(25) (1977) 2340–2361
11. Abrash, H.I.: Studies concerning affinity. Journal of Chemical Education **63** (1986) 1044–1047
12. Meyer, T., Schreckling, D., Tschudin, C., Yamamoto, L.: Robustness to code and data deletion in autocatalytic quines. Volume 5410 of Lecture Notes in Bioinformatics. Springer (2008) 20–40