Code Regulation in Open Ended Evolution

Lidia Yamamoto

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

Abstract. We explore a homeostatic approach to program execution in computer systems: the "concentration" of computation services is regulated according to their fitness. The goal is to obtain a self-healing effect so that the system can resist harmful mutations that could happen during on-line evolution. We present a model in which alternative program variants are stored in a repository representing the organism's "genotype". Positive feedback signals allow code in the repository to be *expressed* (in analogy to gene expression in biology), meaning that it is injected into a reaction vessel (execution environment) where it is executed and evaluated. Since execution is equivalent to a chemical reaction, the program is consumed in the process, therefore needs more feedback in order to be re-expressed. This leads to services that constantly regulate themselves to a stable condition given by the fitness feedback received from the users or the environment. We present initial experiments using this model, implemented using a chemical computing language.

1 Introduction

In a world where computing devices are becoming ubiquitous, it is highly desirable to have systems that are able to operate continuously, adapting to the environment and tracking changing goals. Ideally, such constant self-optimisation should occur not only via hard-wired adaptive algorithms but also through evolution of new functionality without human intervention.

With open ended evolution the system evolves in the environment where it is used. Its fitness must be evaluated online during operation, as opposed to a a well-protected laboratory setting as the one used in off-line evolution. It is therefore particularly important to minimise the impact of potentially harmful genetic operations (mutation, crossover) on the running system. Classical computation models are ill suited for this problem: the typical sequential execution model leads to brittle programs in which adding, deleting or modifying a single instruction may have catastrophic effects.

We investigate the potential of artificial chemical computing models [1–4] for open ended evolution. In these models, change is a rule rather than an exception. Their high parallelism and multiset support allow programs to be expressed in terms of transformations (chemical reactions) applied to their internal objects (data or the transformation rules themselves, in case of high-order models [5]).

pre-print of paper to appear in:
Proceedings of the 10th European Conference on Genetic Programming,
EuroGP 2007, Springer LNCS, Ebner et al (eds.), Valencia, Spain, April 2007
(poster presentation)

Chemical models have been shown that are intrinsically robust to random execution order [6], instruction deletion [7], inherent noise [8, 9], and so on. They have been pointed as suitable for new, autonomic systems able to run unsupervised [10].

In chemical computing, code and data objects are represented as chemical substances that may occur at given *concentrations* in a reaction vessel. The reaction vessel is often implemented as a *multiset*, where elements may occur more than once. The higher the concentration of a given element type, the higher the probability that it participates in a chemical reaction, i.e. that it gets executed (potentially in combination with other elements) to produce a result.

Several reaction vessels might be composed in a hierarchical or recursive way [1, 11] and vessels may communicate with each other by exchanging substances among themselves [11]. Substances react with each other according to specified reaction rules, and the result of the reaction goes to the multiset as well. Substances may be consumed in the reaction, or may act as enzymes for reactions without being consumed in the process.

Computations proceed as chemical reactions, leading to new substances that in turn react with others, and so on, potentially forming large reaction networks. These networks must be regulated to produce the desired result, and have been shown exhibit evolutionary ability [12].

When evolving computational chemical reaction networks online in an open ended context, it is important to ensure that the resulting program can be safely executed, or that any undesirable effects of the execution can be reverted. For this purpose we can take inspiration from biological evolution, which has endowed organisms with several protection mechanisms that ensure viable offspring with high probability. These include redundancy mechanisms such as redundant DNA to protein encoding, diploidy, gene duplication; and self-healing mechanisms such as the regulatory DNA repair cycle, in which proteins respond to DNA damage signals by triggering DNA repair mechanisms, or apoptosis (programmed cell death) when DNA repair is not possible.

We are currently exploring program regulation mechanisms, in which the concentrations of computation objects (at several levels of granularity, such as instructions, modules, programs, or higher level services) are regulated to produce a self-healing effect that resists harmful mutations that could happen during on-line evolution. Fitness evaluation and selection of suitable organisms form integral parts of such regulation cycles. Alternative program variants are stored in a repository which represents the organism's genotype. Code in the repository is *expressed* (in analogy to gene expression in biology) at regular intervals, meaning that it is injected into an execution environment where it is executed and evaluated. Since execution is equivalent to a chemical reaction, the program is consumed in the process.

Code expression happens in response to activation signals coming from the execution environment. These signals are generated in response to *fitness reward* signals coming from the applications evaluating the execution of the program with respect to its capacity to provide the intended service. On the other hand,

programs that are not suitable are evaluated with a low fitness, leading to a *punish* signal that travels to the repository to activate an inhibition rule which prevents the corresponding code to be re-expressed.

With such a mechanism, programs that have a higher fitness are expressed more often, increasing their concentration in the multiset, and therefore end up with a higher probability of being chosen for running.

Contrary to a classical approach where the presence or absence of a program, instruction or file is a binary variable, in such a chemical model objects are present at given concentrations which might increase or decrease in time. When the concentration reaches zero the object disappears from the system. This abstraction lends itself to *homeostatic* computer systems that constantly regulate themselves to a stable condition given by the fitness feedback received from the users or the environment. These systems would be able to adjust to changing requirements or conditions represented by a change in fitness, since they would be constantly seeking a fitness reward to survive.

This paper is structured as follows: Section 2 reviews the current state of the art, and section 3 presents our code regulation approach, with report on experiments in section 4.

2 State of the Art and Related Work

The term *chemical computing* refers to two distinct areas [1, 3]: natural and artificial chemical computing. The first one uses real molecules and chemistry knowledge to build computational devices, e.g. in molecular/DNA computing. The second one derives computation models inspired by chemistry, which nevertheless run on traditional computers. This paper relates to the second area only. In this section we give a brief overview of some of the main artificial chemical computing models and their relation to evolutionary computing.

Artificial chemical computing models have been applied to diverse fields ranging from basic algorithms, to image processing applications, operating systems, compilers, dynamic software reconfiguration, [13], multi-agent systems, distributed computing, and, more recently, robotics [8], grid computing [14], and autonomic computing [15].

In *Gamma* [5] computations are modelled as interactions among atomic values that "float" freely in a chemical solution. These values are represented as elements in a *multiset*, an unordered set within which elements may occur more than once. The number of occurrences of a given element within the multiset is called the *multiplicity* of the element. The multiset contains the data to be processed as well as reaction rules of the form condition-action. Computations replace elements satisfying the condition by those specified in the action. A computation terminates when no more chemical reactions can take place. More recently, γ-calculus has been introduced as an extension of the original Gamma model to a higher-order calculus [5] in which rules are part of the multiset, such that they can also be transformed.

In *Membrane Computing* [1, 11] computations (chemical reactions among *objects*) occur inside a cell-like *membrane structure*. Membranes can be recursively nested. As in Gamma, objects are represented as elements of a multiset. They can be transformed into other objects and can cross membranes.

In *Artificial Chemistries* [2] computations are modelled as chemical reaction networks which can be represented as bi-partite graphs with substrates and reaction rules as nodes. Contrary to Gamma and Membrane Computing, Artificial Chemistries do not focus on specific programming aspects, but rather on the emergent large-scale effects of the interactions among network elements. Chemical organisation theory [16] is used to bridge the gap between the microscopic behaviour at the code (reaction rule) level and the macroscopic behaviour of the system as a whole.

Artificial Regulatory Networks have been shown to model the biological regulatory mechanisms in both natural [9] and artificial systems [12]. In [12] a regulatory network is represented with a genotype/phenotype binary encoding in which genes express proteins, which in turn control the expression of genes, unleashing large reaction networks that evolve by gene duplication and mutations. These networks are able to compute functions, such as a sigmoid and a decaying exponential.

An *Algorithmic Chemistry* [6, 4] is a reaction vessel in which instructions are executed in random order. In [6] the power of genetic programming (GP) applied to an algorithmic chemistry on evolving solutions to specific problems is shown. The authors point out the importance of the concentration of instructions, rather than their sequence. They start from a nearly unpredictable system in which execution of instructions at a random order leads to a random program output. This system is set to evolve by GP, including crossover and mutation of instructions placed in registers, obtaining at the end a highly reproducible output in spite of the random execution order.

Further examples of evolution using artificial chemical systems can be cited, such as the evolution of a robotic control system by GP [8], study of the emergence of evolution using organisation theory [17], evolution of artificial biochemical signalling networks able to compute functions [18], and so on. While these approaches show that chemical and evolutionary systems reinforce each other's potentials, the problem of open-ended general computing in these systems remains a challenging one.

3 Code Regulation

We have developed a code regulation system based on the fraglets chemical language [19]. An interpreter for this language is freely available for download at [20]. The language has a single structure, called a "fraglet" or "computation fragment". A fraglet is a string of symbols $[\, s_1 : s_2 : \ldots : s_n \,]k$ that may encode data, reaction rules involving two fraglets, or transformations of a single fraglet. A suffix counter k indicates the multiplicity of the fraglet in the reaction vessel.

The fraglet language has been designed for network protocols. For this purpose the rule processing engine is based on a "tag matching" system in which fraglets are processed according to their head symbol, which is consumed in the process. This is similar to the way protocol headers in an incoming data packet are processed, consumed and then moved to a higher layer of a protocol stack.

Details on the fraglet language and instruction set can be found in [19]. For our purpose, only one type of rule is important: the reaction rule, called a *match*, or a *matchp* in its persistent variant. A *match* rule has the form $[match : s : tail_1]$, and when meeting a fraglet of the form $[s : tail_2]$ this results in a chemical reaction with product $[tail_1 : tail_2]$, i.e. the two fraglets are concatenated after eliminating their matching heads. The persistent variant *matchp* works in the same way, except that the original *matchp* rule is not consumed during the reaction. Using these rules and other simple transformation rules that manipulate fraglet strings, it is shown that communication protocols and other programs can be implemented [7, 19]. The common point among all the programs shown in [7, 19] is that *matchp* rules are dominant in the code, which makes it rather static as opposed to a real dynamic chemical system in which transformations happen most of the time.

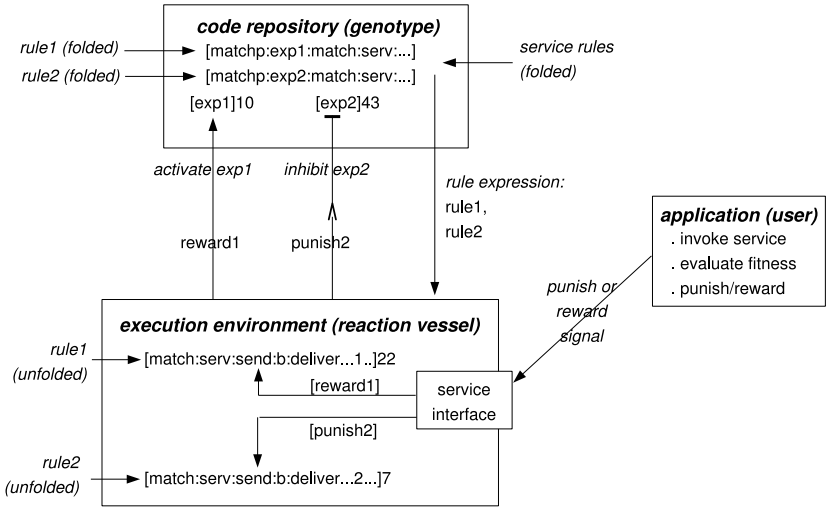

Fig. 1. Code Regulation Scheme

In contrast, we seek a solution in which the concentration of running code is controlled by activating/inhibiting factors and therefore not fixed in advance. We rely solely on the *match* rule and its multiplicity counter for this purpose. These rules are expressed at regular intervals based on the fitness feedback signals

received as a result on an execution that is assumed to provide a given service that can be evaluated during runtime.

Fig. 1 illustrates our code regulation scheme. Rules written as fraglets are stored in a code repository. The repository is also modelled as a reaction vessel, but counts on a persistent memory, so rules stored there are of the form *matchp*. Actually these rules are guarded by an exp_n rule expression factor, where n is the rule number. Rules in this form are said to be "folded". They can only be activated (expressed) by the proper $[exp_n]$ signal, also in the form of a fraglet. The existence of $[exp_n]$ in the repository triggers the corresponding $[matchp : exp_n]$ reaction that consumes one unit of $[exp_n]$ and produces a copy of the rule (free from the $[exp_n]$ guard, or "unfolded") which is injected into the reaction vessel. The rule has the form $[match : serv]$ where $serv$ is the tag corresponding to the service that the rule implements. Examples of services could be to compute a function, or to transmit some data reliably to a destination.

The control mechanism consists then in regulating the amount ("concentration") of $[exp_n]$ in the repository according to the reward and punish signals received during the execution of the corresponding rule. Once inside the execution environment, the rule is activated when the service it performs is invoked by an application by injecting a request of the form $[serv : \ldots parameters \ldots]$. Note that the matching tag ($serv$ in the example of Fig. 1) for both rules is identical, since they are alternative implementations of the same service. These alternative implementations compete for client invocations in the form of fraglets with the matching header tag $[serv : \ldots]$. The proportion of competing $[match : serv : \ldots]$ rules, given by their multiplicity counter, determines the probability of the match reaction to occur involving that rule.

Each rule executed is then either rewarded or punished according to the fitness feedback received from the application. Note that the application is unaware of which of the alternative rules executed the service; it sees only the service as a whole. A service interface is then in charge of translating these generic punish or reward signals to specific signals for the rule that actually performed the service. When rule n is punished, an inhibition signal in the form of a $[punish_n]$ fraglet is issued to the repository, which translates it to a $[match : exp_n]$ fraglet; the later cancels out one unit of exp_n expression signal for rule n. When rewarded, an activation signal $[reward_n]$ is issued for the rule, which translates directly to one or more $[exp_n]$ guards, that will then allow the rule to be re-expressed.

This cycle may continue indefinitely. Note however that the reward/punish signals are only injected by the application, so if there is no demand for a given service, the concentration of its rules will not change in steady state. A certain initial concentration of $[exp_n]$ may be present at the repository, which may cause a few rules to be expressed without demand at the beginning, in order to kick off the cycle; after that, the cycle is completely regulated by the feedback from the application. Since an unfolded rule has the form $[match : serv]$, it is consumed after providing the service, so the only chance for it to keep being executed regularly is to provide a good service that will be properly rewarded.

4 Experiments

We have performed some simulation experiments for code regulation on the fraglets platform. We used the CDP (Confirmed Delivery Protocol) implementation provided in the fraglets package [20] and enhanced it with the regulatory mechanism explained in the previous section.

CDP is a very elementary transport protocol that gets a payload from the application and transmits it to a given destination node in the network. We simulate then a network with two nodes, where the application consists of a data source (co-located with the code repository) and a data sink. Fitness evaluation is performed by the data source: it issues a reward unit upon confirmation of correct delivery of its payload, and a punishment unit when an error occurs or when no confirmation is received.

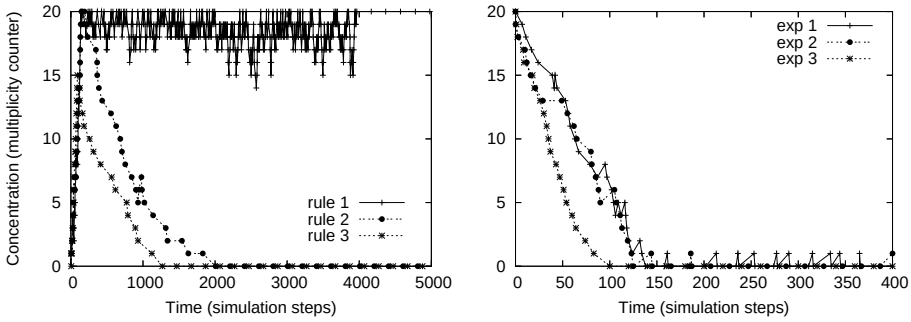

Fig. 2. Concentration of rules expressed to the reaction vessel (left) versus concentration of the corresponding expression signals (right).

Two experiments are shown. In each of them, three rules compete to provide the CDP service to the user: $rule_1$ is an always-good rule (i.e. transmits all the packets faithfully), $rule_3$ is always bad (never transmits anything), and $rule_2$ lies in between: it has a 50% probability of successful transmission.

In the first experiment, each reward for correct execution maps to an increase of a single unit of $[exp_n]$ in the code repository. Conversely, each punishment causes the decrease of one $[exp_n]$ unit, because a $[punish_n]$ fraglet translates into a $[match : exp_n]$ fraglet which eats up one unit of $[exp_n]$. If a rule is good all the time, it will only receive rewards, so each fired rule will lead to an extra unit of $[exp_n]$ being produced, which allows the rule to be re-expressed. On the other hand, expressing a rule consumes one unit of $[exp_n]$, so the concentration of a good rule does not increase in time. So in this experiment, good behaviour is just a way to stay alive in the system, one does not really get rewarded for it. Now if a rule is always bad, then it is punished with the decrease of one $[exp_n]$; on top of the one that had been used for its expression, it is then expected that the concentration of bad rules will rapidly decrease with successive punishments.

Fig. 2 shows the results of the first experiment. The repository is initialised with 20 $[exp_n]$ signals for each rule. The left side shows that the first rule reaches a fixed average concentration (with some oscillation due to the dynamic flow of signals) after a startup period. This concentration is equivalent to the amount of $[exp_1]$ signals that were initially present in the repository. The concentration of the third rule rapidly decreases as the reservoir of initial $[exp_3]$ gets depleted. The second rule lies in between, as expected. The relation between $[exp_n]$ control signals and expressed rules can be seen by comparing both plots on Fig. 2. For readability, the right side shows the concentration of $[exp_n]$ signals for the initial part of the simulation only, since during the remaining simulation time these concentrations do not significantly change.

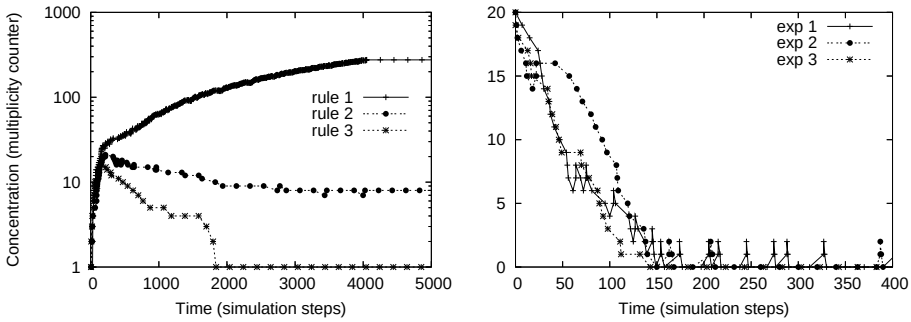

Fig. 3. Concentration of rules and activation signals, second experiment

In the second experiment, every reward for a well executed service triggers two activation signals $[exp_n]$, so the rule is re-expressed (to compensate for its consumption during execution) and at the same time its concentration is allowed to increase (due to an excess of activation signals coming from the rewards). This experiment simulates the situation in which the system provides an incentive for good services to proliferate in the system, and at the same time is able to kill bad ones. Fig. 3 shows the results of the experiment. We can now see that the concentration of $rule_1$ increases as long as it remains in use. Around $t = 4000$ simulation steps the rule is invoked for the last time, so the concentration of $rule_1$ stops growing. Rule 3 is eliminated early as before, while rule 2 now benefits from the extra reward to stay longer in the system, so after the service demand stops there are still around ten rules of type $rule_2$ in the vessel, which remain there in the absence of further control signals.

4.1 Discussion

Note that rules might not be discarded (reach a zero concentration) even if their behaviour is not ideal. This happens when feedback ceases, as with $rule_2$ in the second experiment. This is not necessarily bad, since these "dormant" rules

might represent a useful source of genetic variability which may be helpful for evolution. When conditions change, these rules might be reactivated and become useful in another context, where different service requirements or different environment conditions lead to different performance and resulting fitness.

Currently there is no mechanism to "destroy" rules once they are expressed. The only way to consume a rule is to fire (use) it. It is therefore difficult to punish programs not for low performance, but for lack of demand. One possibility would be to introduce an "aging" mechanism, so that rules that are not invoked for a long time "decay". This would be a useful mechanism to deprecate obsolete software. However, one concern is how to regulate this ageing mechanism itself: how slow or how fast should a service age? if it ages too quickly, the service becomes prematurely unavailable, whereas if it ages too slowly the system gets polluted by obsolete pieces of software (note that this is the situation today, as this "aging" process is controlled by humans in an ad hoc fashion).

The experiments shown are very simple, and intended to shed light on the chemical programming mechanism for code regulation. The next step is to extend them to complex regulation networks in which services are composed of numerous interconnected modules, each of which must be evaluated and selected. For this purpose we consider applying concepts from artificial regulatory networks [9, 12] and chemical organisation theory [16] to fraglet code regulation.

5 Conclusions and Outlook

We have shown a regulation mechanism intended to provide resilience to open ended evolution. This is only one building block for such evolutionary process. The next step is then to show how evolution would actually happen in this context. We believe that the problem of discontinuity of the search space in genetic programming (a small change in a program may cause a big leap in fitness) could be minimised by allowing program or instruction concentrations to be expressed and controlled, instead of binary presence/absence of a given program. This may lead to more stable and predictable transformations in which a small change in a service implementation (which is made of variable concentrations of competing varieties) could lead to a comparably small leap in fitness from the point of view of the user. This cannot be easily achieved with conventional programming languages in which changing a single line of code in an otherwise perfectly working program may cause it to crash, loop or destroy resources. Chemical systems with regulated computations could lead to a much more elastic way to handle exceptions and therefore to a robust way to support online evolutionary computation for systems that may be made to run forever uninterrupted.

References

1. Calude, C.S., Paun, G.: Computing with Cells and Atoms: An Introduction to Quantum, DNA and Membrane Computing. Taylor & Francis (2001)

2. Dittrich, P., Ziegler, J., Banzhaf, W.: Artificial Chemistries – A Review. Artificial Life **7**(3) (2001) 225–275
3. Dittrich, P.: Chemical Computing. In: Unconventional Programming Paradigms (UPP 2004), Springer LNCS 3566. (2005) 19–32
4. W.G.Lasarczyk, C., Banzhaf, W.: An Algorithmic Chemistry for Genetic Programming. In: Proc. 8th European Conference on Genetic Programming. M. Keijzer, A. Tettamanzi, P. Collet, M. Tomassini, J. van Hemert (Eds.) Springer LNCS 3447, Lausanne, Switzerland (2005) 1–129
5. 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).
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. Tschudin, C., Yamamoto, L.: A Metabolic Approach to Protocol Resilience. In: Proc. 1st Workshop on Autonomic Communication (WAC). Springer LNCS 3457, Berlin, Germany (2004) 190–205
8. Ziegler, J., Banzhaf, W.: Evolving Control Metabolisms for a Robot. Artificial Life **7**(2) (2001) 171–190
9. 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. Springer LNCS 3905, Budapest, Hungary (2006) 290–299
10. Banâtre, J.P., Radenac, Y., Fradet, P.: Chemical Specification of Autonomic Systems. In: Proc 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE'04). (2004) 72–79
11. Paun, G.: Computing with Membranes. Journal of Computer and System Sciences **61**(1) (2000) 108–143
12. Kuo, P., Banzhaf, W., Leier, A.: Network topology and the evolution of dynamics in an artificial genetic regulatory network model created by whole genome duplication and divergence. Biosystems **85** (2006) 177–200
13. Wermelinger, M.A.: Specification of Software Architecture Reconfiguration. PhD dissertation, Universidade Nova de Lisboa, Lisbon, Portugal (1999)
14. Banâtre, J.P., Fradet, P., Radenac, Y.: Towards Grid Chemical Coordination. In: Proceedings of Symposium on Applied Computing (SAC). (2006) (short paper).
15. Banâtre, J.P., Radenac, Y., Fradet, P.: Chemical specification of autonomic systems. In: Proc 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE'04). (2004) 72–79
16. Matsumaru, N., Centler, F., di Fenizio, P.S., Dittrich, P.: Chemical Organization Theory as a Theoretical Base for Chemical Computing. International Journal of Unconventional Computing (in print) (2006)
17. Matsumaru, N., di Fenizio, P.S., Centler, F., Dittrich, P.: On the Evolution of Chemical Organizations. In: Proc. 7th German Workshop on Artificial Life. (2006) 135–146
18. Deckard, A., Sauro, H.M.: Preliminary Studies on the In Silico Evolution of Biochemical Networks. ChemBioChem **5**(10) (2004) 1423–1431
19. Yamamoto, L., Tschudin, C.: Experiments on the Automatic Evolution of Protocols using Genetic Programming. In: Proc. 2nd Workshop on Autonomic Communication (WAC), Athens, Greece (2005) 13–28
20. Fraglets Home Page: http://www.fraglets.net/ (2005)