

# Higher-order Chemical Programming Style

J.-P. Banâtre<sup>1</sup>, P. Fradet<sup>2</sup> and Y. Radenac<sup>1</sup> [19th July 2004]

<sup>1</sup> IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France  
(jbanatre,yradenac)@irisa.fr

<sup>2</sup> INRIA Rhône-Alpes, 655 avenue de l'Europe, 38330 Montbonnot, France  
Pascal.Fradet@inria.fr

**Abstract.** The chemical reaction metaphor describes computation in terms of a chemical solution in which molecules interact freely according to reaction rules. Chemical solutions are represented by multisets of elements and computation proceeds by consuming and producing new elements according to reaction conditions and transformation rules. The chemical programming style allows to write many programs in a very elegant way. We go one step further by extending the model so that rewrite rules are themselves molecules. This higher-order extension leads to a programming style where the implementation of new features amounts to adding new active molecules in the solution representing the system.

## 1 Introduction

The chemical reaction metaphor has been discussed in various occasions in the literature. This metaphor describes computation in terms of a chemical solution in which molecules (representing data) interact freely according to reaction rules. Chemical solutions are represented by multisets. Computation proceeds by rewritings of the multiset which consume and produce new elements according to reaction conditions and transformation rules.

To the best of our knowledge, the Gamma formalism was the first “chemical model of computation” proposed as early as in 1986 [1] and later extended in [2]. A Gamma program is a collection of reaction rules acting on a multiset of basic elements. A reaction rule is made of a condition and an action. Execution proceeds by replacing elements satisfying the reaction condition by the elements specified by the action. The result of a Gamma program is obtained when a stable state is reached, that is to say, when no reaction can take place any more. Figure 1 gives three short examples illustrating the Gamma style of programming. The

$$\begin{aligned}max &= \mathbf{replace} \ x, y \ \mathbf{by} \ x \ \mathbf{if} \ x > y \\primes &= \mathbf{replace} \ x, y \ \mathbf{by} \ y \ \mathbf{if} \ multiple(x, y) \\maj &= \mathbf{replace} \ x, y \ \mathbf{by} \ \{\} \ \mathbf{if} \ x \neq y\end{aligned}$$

**Fig. 1.** Examples of Gamma programs

reaction *max* computes the maximum element of a non empty set. The reaction replaces any couple of elements  $x$  and  $y$  such that  $x > y$  by  $x$ . This process goes on till a stable state is reached, that is to say, when only the maximum element remains. The reaction *primes* computes the prime numbers lower or equal to a given number  $N$  when applied to the multiset of all numbers between 2 and  $N$  (*multiple*( $x, y$ ) is true if and only if  $x$  is multiple of  $y$ ). The majority element of a multiset is an element which occurs more than  $card(M)/2$  times in the multiset. Assuming that such an element exists, the reaction *maj* yields a multiset which only contains instances of the majority element just by removing pairs of distinct elements. Let us emphasize the conciseness and elegance of these programs. Nothing had to be said about the order of evaluation of the reactions. If several disjoint pairs of elements satisfy the condition, the reactions can be performed in parallel.

Gamma makes it possible to express programs without artificial sequentiality. By artificial, we mean sequentiality only imposed by the computation model and unrelated to the logic of the program. This allows the programmer to describe programs in a very abstract way. In some sense, one can say that Gamma programs express the very idea of an algorithm without any unnecessary linguistic idiosyncrasies. The interested reader may find in [2] a long series of examples (string processing problems, graph problems, geometry problems, ...) illustrating the Gamma style of programming and in [3] a review of contributions related to the chemical reaction model.

This article presents a higher-order extension of the Gamma model where all the computing units are considered as molecules reacting in a solution. In particular, reaction rules are molecules which can react or be manipulated as any other molecules. In Section 2, we exhibit a minimal higher-order chemical calculus, called the  $\gamma$ -calculus, which expresses the very essence of chemical models. This calculus is then enriched with conditional reactions and the possibility of rewriting atomically several molecules. This model suggests a programming style where the implementation of new features amounts to adding new active molecules in the solution representing the system. Section 3 illustrates the characteristics of the model through the example of an autonomic mail system. Section 4 concludes and suggests several research directions.

## 2 A minimal chemical calculus

In this section, we introduce a higher-order calculus, the  $\gamma_0$ -calculus [4], that can be seen as a formal and minimal basis for the chemical paradigm (in much the same way as the  $\lambda$ -calculus is the formal basis of the functional paradigm).

### 2.1 Syntax and semantics

The fundamental data structure of the  $\gamma_0$ -calculus is the multiset. Computation can be seen either intuitively, as chemical reactions of elements agitated by Brownian motion, or formally, as higher-order associative and commutative (AC)

rewritings of multisets. The syntax of  $\gamma_0$ -terms (also called *molecules*) is given in Figure 2. A  $\gamma$ -abstraction is a reactive molecule which consumes a molecule (its

$$\begin{array}{l}
 M ::= x \quad ; \textit{variable} \\
 \quad | \quad \gamma\langle x \rangle.M \quad ; \textit{\gamma-abstraction} \\
 \quad | \quad M_1, M_2 \quad ; \textit{multiset} \\
 \quad | \quad \langle M \rangle \quad ; \textit{solution}
 \end{array}$$

**Fig. 2.** Syntax of  $\gamma_0$ -molecules

argument) and produces a new one (its body). Molecules are composed using the AC multiset constructor “,”. A solution encapsulates molecules (*e.g.*, multiset) and keeps them separate. It serves to control and isolate reactions.

The  $\gamma_0$ -calculus bears clear similarities with the  $\lambda$ -calculus. They both rely on the notions of (free and bound) variable, abstraction and application. A  $\lambda$ -abstraction and a  $\gamma$ -abstraction both specify a higher-order rewrite rule. However,  $\lambda$ -terms are tree-like whereas the AC nature of the application operator “,” makes  $\gamma_0$ -terms multiset-like. Associativity and commutativity formalizes Brownian motion and make the notion of solution necessary, if only to distinguish between a function and its argument.

The conversion rules and the reduction rule of the  $\gamma_0$ -calculus are gathered in Figure 3. Chemical reactions are represented by a single rewrite rule, the  $\gamma$ -

$$\begin{array}{l}
 (\gamma\langle x \rangle.M), \langle N \rangle \longrightarrow_{\gamma} M[x := N] \quad \text{if } \textit{Inert}(N) \vee \textit{Hidden}(x, M) \quad ; \textit{\gamma-reduction} \\
 \gamma\langle x \rangle.M \quad \equiv \quad \gamma\langle y \rangle.M[x := y] \quad \text{with } y \textit{ fresh} \quad ; \textit{\alpha-conversion} \\
 M_1, M_2 \quad \equiv \quad M_2, M_1 \quad ; \textit{commutativity} \\
 M_1, (M_2, M_3) \quad \equiv \quad (M_1, M_2), M_3 \quad ; \textit{associativity}
 \end{array}$$

**Fig. 3.** Rules of the  $\gamma_0$ -calculus

reduction, which applies a  $\gamma$ -abstraction to a solution. A molecule  $(\gamma\langle x \rangle.M), \langle N \rangle$  can be reduced only if:

- $\textit{Inert}(N)$ : the content  $N$  of the solution argument is a closed term made exclusively of  $\gamma$ -abstractions or exclusively of solutions (which may be active),
- or**  $\textit{Hidden}(x, M)$ : the variable  $x$  occurs in  $M$  only as  $\langle x \rangle$ . Therefore  $\langle N \rangle$  can be active since no access is done to its contents.

So, a molecule can be extracted from its enclosing solution only when it has reached an inert state. This is an important restriction that permits the ordering of rewritings. Without this restriction, the contents of a solution could be extracted in any state and the solution construct would lose its purpose. Reactions can occur in parallel as long as they apply to disjoint sub-terms. A molecule is in normal form if all its molecules are inert.

Consider, for example, the following molecules:

$$w \equiv \gamma\langle x \rangle.x, \langle x \rangle \quad \Omega \equiv w, \langle w \rangle \quad I \equiv \gamma\langle x \rangle.\langle x \rangle$$

Clearly,  $\Omega$  is an always active (non terminating) molecule and  $I$  an inert molecule (the identity function in normal form). The molecule  $\langle \Omega \rangle, \langle I \rangle, \gamma\langle x \rangle.\gamma\langle y \rangle.x$  reduces as follows:

$$\langle \Omega \rangle, \langle I \rangle, \gamma\langle x \rangle.\gamma\langle y \rangle.x \longrightarrow \langle \Omega \rangle, \gamma\langle y \rangle.I \longrightarrow I$$

The first reduction is the only one possible: the  $\gamma$ -abstraction extracts  $x$  from its solution and  $\langle I \rangle$  is the only inert molecule ( $Inert(I) \wedge \neg Hidden(x, \gamma\langle y \rangle.x)$ ). The second reduction is possible only because the active solution  $\langle \Omega \rangle$  is not extracted but removed ( $\neg Inert(\Omega) \wedge Hidden(y, I)$ )

## 2.2 Two fundamental extensions

The  $\gamma_0$ -calculus is a quite expressive higher-order calculus. However, compared to the original Gamma [2] and other chemical models [5,6], it lacks two fundamental features:

- *Reaction condition.* In Gamma, reactions are guarded by a condition that must be fulfilled in order to apply them. Compared to  $\gamma_0$  where inertia and termination are described syntactically, conditional reactions give these notions a semantic nature.
- *Atomic capture.* In Gamma, any fixed number of elements can take part in a reaction. Compared to a  $\gamma_0$ -abstraction which reacts with one element at a time, a  $n$ -ary reaction takes atomically  $n$  elements which cannot take part in any other reaction at the same time.

These two extensions are orthogonal and enhance greatly the expressivity of chemical calculi. So from now, we consider the  $\gamma$ -calculus extended with booleans, integers, arithmetic and booleans operators, tuples (written  $x_1: \dots :x_n$ ) and the possibility of naming molecules ( $ident = M$ ). Furthermore,  $\gamma$ -abstractions (also called *active molecules*) can react according to a condition and can extract elements using pattern-matching. The syntax of  $\gamma$ -abstractions is extended to:

$$\gamma P[C].M$$

where  $M$  is the action,  $C$  is the reaction condition and  $P$  a pattern extracting the elements participating in the reaction. Patterns have the following syntax:

$$P ::= x \mid \omega \mid ident = P \mid P, P \mid \langle P \rangle$$

where

- variables ( $x$ ) match basic elements (integers, booleans, tuples, ...),
- $\omega$  is a named wild card that matches any molecule (even the empty one),
- $ident = P$  matches any molecule  $m$  named  $ident$  which matches  $P$ ,
- $P_1, P_2$  matches any molecule  $(m_1, m_2)$  such that  $m_1$  matches  $P_1$  and  $m_2$  matches  $P_2$ ,
- $\langle P \rangle$  matches any solution  $\langle m \rangle$  such that  $m$  matches  $P$ .

For example, the pattern  $Sol = \langle x, y, \omega \rangle$  matches any solution named  $Sol$  containing at least two basic elements named  $x$  and  $y$ . The rest of the solution (that may be empty) is matched by  $\omega$ .

$\gamma$ -abstractions are one-shot: they are consumed by the reaction. However, many programs are naturally expressed by applying the same reaction an arbitrary number of times. We introduce recursive (or  $n$ -shot)  $\gamma$ -abstractions which are not consumed by the reaction. We denote them by the following syntax:

**replace  $P$  by  $M$  if  $C$**

Such a molecule reacts exactly as  $\gamma P[C].M$  except than it remains after the reaction and can be used as many times as necessary. If needed, they can be removed by another molecule, thanks to the higher-order nature of the language. If the condition  $C$  is **true**, we omit it in the definition of one-shot or  $n$ -shot molecules.

A higher-order Gamma program is an unstable solution of molecules. The execution of that program consists in performing the reactions (modulo A/C) until a stable state is reached (no more reaction can occur).

### 3 Towards an autonomic mail system

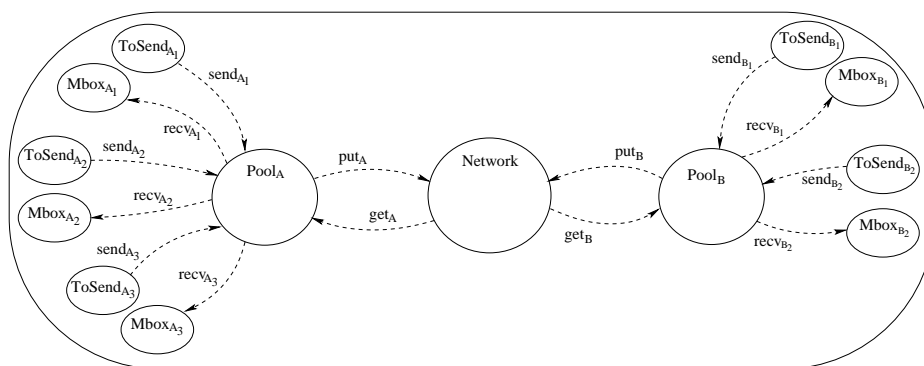
In this section, we describe an autonomic mail system within the Gamma framework. This example illustrate the adequacy of the chemical paradigm to the description of autonomic systems.

#### 3.1 General description: self-organization.

The mail system consists in servers, each one dealing with a particular address domain, and clients sending their messages to their domain server. Servers forward messages addressed to other domains to the network. They also get messages addressed to their domain from the network and direct them to the appropriate clients. The mail system (see Figure 4) is described using several molecules:

- Messages exchanged between clients are represented by basic molecules whose structure is left unspecified. We just assume that relevant information (such as sender’s address, recipient’s address, etc.) can be extracted using appropriate functions (such as *sender*, *recipient*, *senderDomain*, etc.).
- Solutions named  $ToSend_{d_i}$  contain the messages to be sent by the client  $i$  of domain  $d$ .

- Solutions named  $Mbox_{d_i}$  contain the messages received by the client  $i$  of domain  $d$ .
- Solutions named  $Pool_d$  contain the messages that the server of domain  $d$  must take care of.
- The solution named  $Network$  represents the global network interconnecting domains.
- A client  $i$  in domain  $d$  is represented by two active molecules  $send_{d_i}$  and  $rcv_{d_i}$ .
- A server of a domain  $d$  is represented by two active molecules  $put_d$  and  $get_d$ .



**Fig. 4.** Mail system.

Clients send messages by adding them to the pool of messages of their domain. They receive messages from the pool of their domain and store them in their mailbox. The  $send_{d_i}$  molecule sends messages of the client  $i$  (*i.e.*, messages in the  $ToSend_{d_i}$  solution) to the client's domain pool (*i.e.*, the  $Pool_d$  solution). The  $rcv_{d_i}$  molecule places the messages addressed to client  $i$  (*i.e.*, messages in the  $Pool_d$  solution whose recipient is  $i$ ) in the client's mailbox (*i.e.*, the  $Mbox_{d_i}$  solution).

Servers forward messages from their pool to the network. They receive messages from the network and store them in their pool. The  $put_d$  molecule forwards only messages addressed to other domains than  $d$ . The molecule  $get_d$  extracts messages addressed to  $d$  from the network and places them in the pool of domain  $d$ . The system is a solution, named  $MailSystem$ , containing molecules representing clients, messages, pools, servers, mailboxes and the network. Figure 4 represents graphically the solution with five clients grouped into two domains  $A$  and  $B$  and Figure 5 provides the definition of the molecules.

```

senddi = replace ToSenddi = ⟨msg, ωt⟩, Poold = ⟨ωp⟩
           by ToSenddi = ⟨ωt⟩, Poold = ⟨msg, ωp⟩
recvdi = replace Poold = ⟨msg, ωp⟩, Mboxdi = ⟨ωb⟩
           by Poold = ⟨ωp⟩, Mboxdi = ⟨msg, ωb⟩
           if recipient(msg) = i

putd = replace Poold = ⟨msg, ωp⟩, Network = ⟨ωn⟩
        by Poold = ⟨ωp⟩, Network = ⟨msg, ωn⟩
        if recipientDomain(msg) ≠ d

getd = replace Network = ⟨msg, ωn⟩, Poold = ⟨ωp⟩
        by Network = ⟨ωn⟩, Poold = ⟨msg, ωp⟩
        if recipientDomain(msg) = d

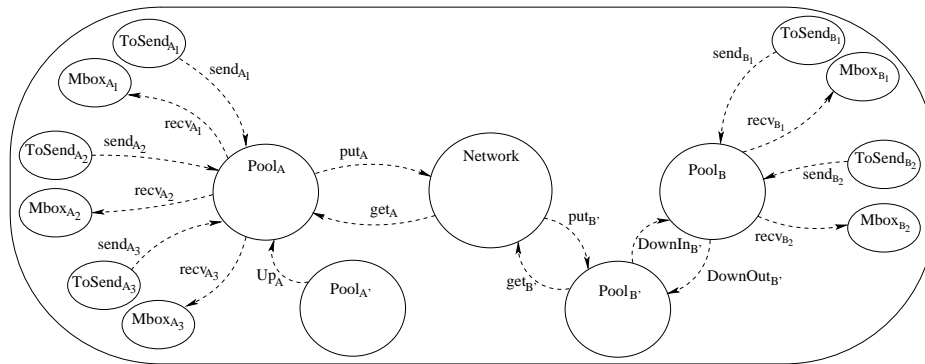
MailSystem = ⟨ sendA1, recvA1, ToSendA1 = ⟨...⟩, MboxA1 = ⟨...⟩,
               sendA2, recvA2, ToSendA2 = ⟨...⟩, MboxA2 = ⟨...⟩,
               sendA3, recvA3, ToSendA3 = ⟨...⟩, MboxA3 = ⟨...⟩,
               putA, getA, PoolA, Network, putB, getB, PoolB,
               sendB1, recvB1, ToSendB1 = ⟨...⟩, MboxB1 = ⟨...⟩,
               sendB2, recvB2, ToSendB2 = ⟨...⟩, MboxB2 = ⟨...⟩
             ⟩

```

**Fig. 5.** Self-organization molecules.

### 3.2 Self-healing.

We now assume that a server may crash. To prevent the mail service from being discontinued, we add an emergency server for each domain (see Figure 6). The



**Fig. 6.** Highly-available mail system.

$$\begin{aligned}
\text{crashServer}_d &= \mathbf{replace} \text{ put}_d, \text{ get}_d, \text{ Up}_d \\
&\quad \mathbf{by} \text{ put}_{d'}, \text{ get}_{d'}, \text{ DownIn}_d, \text{ DownOut}_d \\
&\quad \mathbf{if} \text{ failure}(d) \\
\text{repairServer}_d &= \mathbf{replace} \text{ put}_{d'}, \text{ get}_{d'}, \text{ DownIn}_d, \text{ DownOut}_d \\
&\quad \mathbf{by} \text{ put}_d, \text{ get}_d, \text{ Up}_d \\
&\quad \mathbf{if} \text{ recover}(d) \\
\\
\text{DownOut}_d &= \mathbf{replace} \text{ Pool}_d = \langle \text{msg}, \omega_p \rangle, \text{ Pool}_{d'} = \langle \omega_n \rangle \\
&\quad \mathbf{by} \text{ Pool}_d = \langle \omega_p \rangle, \text{ Pool}_{d'} = \langle \text{msg}, \omega_n \rangle \\
&\quad \mathbf{if} \text{ domain}(\text{msg}) \neq d \\
\text{DownIn}_d &= \mathbf{replace} \text{ Pool}_d = \langle \omega_p \rangle, \text{ Pool}_{d'} = \langle \text{msg}, \omega_n \rangle \\
&\quad \mathbf{by} \text{ Pool}_d = \langle \text{msg}, \omega_p \rangle, \text{ Pool}_{d'} = \langle \omega_n \rangle \\
&\quad \mathbf{if} \text{ domain}(\text{msg}) = d \\
\\
\text{Up}_d &= \mathbf{replace} \text{ Pool}_{d'} = \langle \text{msg}, \omega_p \rangle, \text{ Pool}_d = \langle \omega_n \rangle \\
&\quad \mathbf{by} \text{ Pool}_{d'} = \langle \omega_p \rangle, \text{ Pool}_d = \langle \text{msg}, \omega_n \rangle \\
\\
\text{MailSystem} &= \langle \dots, \text{Up}_A, \text{Up}_B, \text{Pool}'_A, \text{Pool}'_B, \text{crashServer}_A, \text{repairServer}_A, \\
&\quad \text{crashServer}_B, \text{repairServer}_B \rangle
\end{aligned}$$

**Fig. 7.** Self-healing molecules.

emergency servers work with their own pool as usual but are active only when the corresponding main server has crashed. The modeling of a server crash can be done using the reactive molecules described in Figure 7. When a failure occurs, the active molecules representing a main server are replaced by molecules representing the corresponding emergency server. The boolean *failure* denotes a (potentially complex) failure detection mechanism. The inverse reaction *repairServer* represents the recovery of the server.

The two molecules  $\text{Up}_d$  and  $(\text{DownIn}_d, \text{DownOut}_d)$  represent the state of the main server  $d$  in the solution, but they are also active molecules in charge of transferring pending messages from  $\text{Pool}_d$  to  $\text{Pool}_{d'}$ ; then, they may be forwarded by the emergency server.

The molecule  $\text{DownOut}_d$  transfers all messages bound to another domain than  $d$  from the main pool  $\text{Pool}_d$  to the emergency pool  $\text{Pool}_{d'}$ . The molecule  $\text{DownIn}_d$  transfers all messages bound to the domain  $d$  from the emergency pool  $\text{Pool}_{d'}$  to the main pool  $\text{Pool}_d$ .

After a transition from the *Down* state to the *Up* state, it may remain some messages in the emergency pools. So, the molecule  $\text{Up}_d$  brings back all the messages of the emergency pool  $\text{Pool}_{d'}$  into the the main pool  $\text{Pool}_d$  to be then treated by the repaired main server. In our example, self-healing can be implemented by two emergency servers  $A'$  and  $B'$  and boils down to adding the molecules of Figure 7 into the main solution.

Other self-management features have been developed in [7]: self-optimization (by enabling the emergency server and load-balancing messages between it and the main server), self-protection (detect and suppress spams) and self-configuration (managing mobile clients).

Our description should be regarded as a high-level parallel and modular specification. It allows to design and reason about autonomic systems at an appropriate level of abstraction. Let us emphasize the beauty of the resulting programs which rely essentially on the higher-order and chemical nature of Gamma. A direct implementation of this program is likely to be quite inefficient and further refinements are needed; this is another exciting research direction, not tackled here.

## 4 Conclusion

We have presented a higher-order multiset transformation language which can be described using a chemical reaction metaphor. The higher-order property of our model makes it much more powerful and expressive than the original Gamma [2] or than the Linda language as described in [8]. In this article, we have shown the fundamental features of the chemical programming paradigm. The  $\gamma_0$ -calculus embodies the essential characteristics (AC multiset rewritings) in only four syntax rules. This minimal calculus has been shown to be expressive enough to express the  $\lambda$ -calculus and a large class of non-deterministic programs [4]. However, in order to come close to a real chemical language, two extensions must be considered: reaction conditions and atomic capture. With appropriate syntactic sugar (recursion, constants, operators, pattern-matching, etc.), the extended calculus can easily express most of the existing chemical languages.

In this higher-order model, molecules representing reaction rules can be seen as catalysts that perform computations and implements new features. This programming style has been illustrated by the example of an autonomic mail system described as molecules and transformation rules. These rules may apply as soon as a predefined condition holds without external intervention. In other words, the system configures and manages itself to face predefined situations. Our chemical mail system shows that our approach is well-suited to the high-level description of autonomic systems. Reaction rules exhibit the essence of “autonomy” without going into useless details too early in the development process.

An interesting research direction is to take advantage of these high-level descriptions to carry out proofs of properties of autonomic systems (in the same spirit as [9]). For example, “not losing any messages” would be an important property to prove for our mail system. Another direction would be to extend our language to prevent clumsy encodings (*e.g.*, using advanced data structures and others high-level facilities).

## References

1. Banâtre, J.P., Le Métayer, D.: A new computational model and its discipline of programming. Technical Report RR0566, INRIA (1986)

2. Banâtre, J.P., Le Métayer, D.: Programming by multiset transformation. *Communications of the ACM (CACM)* **36** (1993) 98–111
3. Banâtre, J.P., Fradet, P., Le Métayer, D.: Gamma and the chemical reaction model: Fifteen years after. In: *Multiset Processing*. Volume 2235 of LNCS., Springer-Verlag (2001) 17–44
4. Banâtre, J.P., Fradet, P., Radenac, Y.: Principles of chemical programming. In: *RULE'04*. ENTCS, Elsevier (2004) (to appear).
5. Le Métayer, D.: Higher-order multiset programming. In (AMS), A.M.S., ed.: *Proc. of the DIMACS workshop on specifications of parallel algorithms*. Volume 18 of *Dimacs Series in Discrete Mathematics*. (1994)
6. Păun, G.: Computing with membranes. *Journal of Computer and System Sciences* **61** (2000) 108–143
7. Banâtre, J.P., Fradet, P., Radenac, Y.: Chemical specification of autonomic systems. In: *IASSE'04*. (2004) (to appear).
8. Carriero, N., Gelernter, D.: Linda in Context. *Communications of the ACM* **32** (1989) 444–458
9. Barradas, H.: Systematic derivation of an operating system kernel in Gamma. Phd thesis (in french), University of Rennes, France (1993)