Higher-order Chemical Programming Style

J.-P. Banâtre[1], P. Fradet[2] and Y. Radenac[1]

[1] IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
(jbanatre,yradenac)@irisa.fr
[2] 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 reactions by rewrite rules which consume and produce new elements according to conditions. 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. We illustrate this style by specifying an autonomic mail system with several self-managing properties.

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 anymore.

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

Fig. 1. Examples of Gamma programs

Figure 1 gives three small examples illustrating the style of programming of Gamma. The reaction *max* computes the maximum element of a non empty set. The reaction replaces any couple of elements x and y such that the reaction condition $(x \geq y)$ holds 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 γ-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. The resulting higher-order chemical language 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 our language through the example of an autonomic mail system with several self-managing features. Section 4 concludes and suggests several research directions.

2 A minimal chemical calculus

In this section, we introduce a higher-order calculus, the *γ-calculus* [4], that can be seen as a formal and minimal basis for the chemical paradigm in much the same way as the λ-calculus is the formal basis of the functional paradigm.

2.1 Syntax and semantics

The fundamental data structure of the γ-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), multiset rewritings. The syntax of γ-terms (also called *molecules*) is given in Figure 2. A γ-abstraction is a reactive molecule which consumes a molecule

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

Fig. 2. Syntax of γ-molecules

(its 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 γ-calculus bears clear similarities with the λ-calculus. They both rely on the notions of (free and bound) variable, abstraction and application. A λ-abstraction and a γ-abstraction both specify a higher-order rewrite rule. However, λ-terms are tree-like whereas the AC nature of the application operator "," makes γ-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 γ-calculus are gathered in Figure 3. Chemical reactions are represented by a single rewrite rule, the γ-

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

Fig. 3. Rules of the γ-calculus

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

Inert(N): the content N of the solution argument is a closed term made exclusively of γ-abstractions or exclusively of solutions (which may be active),

or *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.

In order to illustrate γ-reduction, consider the following molecules:

$$\Delta \equiv \gamma\langle x\rangle.x, \langle x\rangle \qquad \Omega \equiv \Delta, \langle\Delta\rangle \qquad I \equiv \gamma\langle x\rangle.\langle x\rangle$$

Clearly, Ω 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 γ-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 ($\neg Inert(\Omega) \wedge Hidden(y, I)$).

Like in the λ-calculus, constants can be defined using basic constructs. For example, booleans and conditionals can be encoded as follows:

$$\begin{aligned} \textbf{true} &\equiv \gamma\langle x\rangle.\gamma\langle y\rangle.x \\ \textbf{false} &\equiv \gamma\langle x\rangle.\gamma\langle y\rangle.y \\ \textbf{if}\, C\, \textbf{then}\, M_1\, \textbf{else}\, M_2 &\equiv \langle\langle C\rangle, \gamma\langle x\rangle.x, \langle M_1\rangle\rangle, \gamma\langle y\rangle.y, \langle M_2\rangle \end{aligned}$$

In the encoding of the conditional, when the molecule C reduces to **true** (resp. **false**) the whole expression reduces to M_1 (resp. M_2). Other standard constructions (pairs, tuples, integers, recursion,. . .) can be encoded as well. Actually, the λ-calculus can easily be encoded within the γ-calculus (see [4] for more details).

In fact, the γ-calculus is more expressive than the λ-calculus since it can also express non-deterministic programs. For example, let A and B two distinct normal forms, then:

$$\begin{array}{ccc} (\gamma\langle x\rangle.\gamma\langle y\rangle.x), \langle A\rangle, \langle B\rangle & \equiv & (\gamma\langle x\rangle.\gamma\langle y\rangle.x), \langle B\rangle, \langle A\rangle \\ \downarrow_\gamma & & \downarrow_\gamma \\ (\gamma\langle y\rangle.A), \langle B\rangle & & (\gamma\langle y\rangle.B), \langle A\rangle \\ \downarrow_\gamma & & \downarrow_\gamma \\ A & \not\equiv & B \end{array}$$

The γ-calculus is not confluent.

2.2 Extensions

The γ-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 γ 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 γ-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, γ-abstractions (also called *active molecules*) can react according to a condition and can extract elements using pattern-matching. Furthermore, we consider the γ-calculus extended with booleans, integers, arithmetic and booleans operators, tuples (written $x_1{:}\ldots{:}x_n$) and the possibility of naming molecules ($ident = M$). The syntax of γ-abstractions is extended to:

$$\gamma P\lfloor C\rfloor.M$$

where M is the action, C is the reaction condition and P a pattern extracting the elements participating in the reaction. If the condition C is **true**, we omit it in the definition of the γ-abstraction.

Patterns have the following syntax:

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

where

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

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

γ-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) γ-abstractions which are not consumed by the reaction. We denote them by the following syntax:

$$\textbf{replace}\ P\ \textbf{by}\ M\ \textbf{if}\ C$$

Such a molecule reacts exactly as $\gamma P\lfloor C\rfloor.M$ except than it remains after the reaction and can be used as many times as necessary. If needed, a reactive molecule can be removed by another molecule, thanks to the higher-order nature of the language.

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 (*i.e.,* no more reaction can occur). A standard Gamma program can be represented in our extended calculus by encoding its reaction rules by n-shot abstractions placed in the multiset.

For example, the Gamma program computing the maximum element of a multiset of integers is represented by a reaction rule (*max* in Figure 1) to be applied to the multiset. In our higher-order model, that rule is considered as a molecule in the solution of integer molecules. Figure 4 illustrates such a solution and its reduction. Like in the original Gamma, the program terminates when no

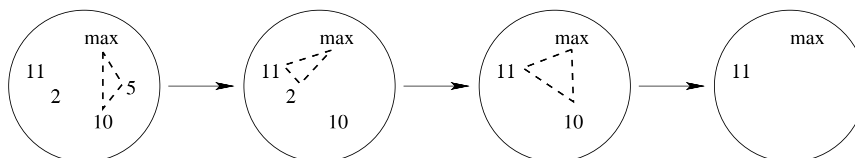

Fig. 4. A possible execution of the program computing the maximum.

more reactions can occur. In our example, the solution becomes inert when only one integer (the maximum) remains.

The following solution computes the greatest common divisor (gcd) of its two integers:

$$\langle \text{init}, \text{gcd}, \text{clean}, 15, 21 \rangle$$

where

$$\begin{aligned} \text{init} &= \gamma(x, y)\lfloor x \geq y \rfloor.x{:}y \\ \text{gcd} &= \textbf{replace}\, x{:}y\, \textbf{by}\, y{:}(x\ mod\ y)\, \textbf{if}\, y \neq 0 \\ \text{clean} &= \gamma(x{:}y, \text{gcd})\lfloor y = 0 \rfloor.x \end{aligned}$$

First, only the abstraction "init" can react. It places the two integers in a pair and disappears (one-shot abstraction). Then, the molecule "gcd" transforms sequentially the pair until the second place is null ($x\ mod\ y$ yields the rest of the division of x by y). Finally, the one-shot abstraction "clean" reacts: it extracts the result (x) from the pair and removes the gcd molecule.

Names can be used to tag any molecule: abstractions, solutions, ... For example, if we name "Gcd" the following solution computing the gcd of two integers:

$$\text{Gcd} = \langle \text{init}, \text{gcd}, \text{clean} \rangle$$

then the abstraction computing the gcd of two parameters can be written:

$$\gamma(\text{Gcd} = \langle \omega \rangle, x, y).\langle \omega, x, y \rangle$$

It builds a solution made of the molecules init, gcd, clean (*i.e.,* Gcd) and the two parameters x and y (assumed to be integers). When the solution becomes inert, only the gcd of x and y remains.

N-shot abstractions are well fitted to express self-management properties. For example, computing the prime numbers up to 5 can be expressed as:

$$\langle \text{primes}, 2, 3, 4, 5\rangle \longrightarrow_\gamma \langle \text{primes}, 2, 3, 5\rangle$$

where *primes* is the reaction of Figure 1. The molecule "primes" is part of the result (stable state). If new integers are added (perturbation), reactions may start again until a new inert solution is reached (new stable state). For example, if we need the prime numbers up to 10, we may just add integers to the previous inert solution:

$$\langle \text{primes}, 2, 3, 4, 5\rangle,\ \gamma\langle x\rangle.\langle x, 6, 7, 8, 9, 10\rangle$$

and the solution will re-stabilize to $\langle \text{primes}, 2, 3, 5, 7\rangle$. The molecule "primes" can be seen as an invariant: it describes the valid inert states (here, set of prime numbers). In the next section, we make use of this property to add several self-management features to a mail system.

3 Towards an autonomic mail system

In this section, we describe an autonomic mail system within our higher-order chemical framework. This example illustrates 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 5) 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 recv_{d_i}.

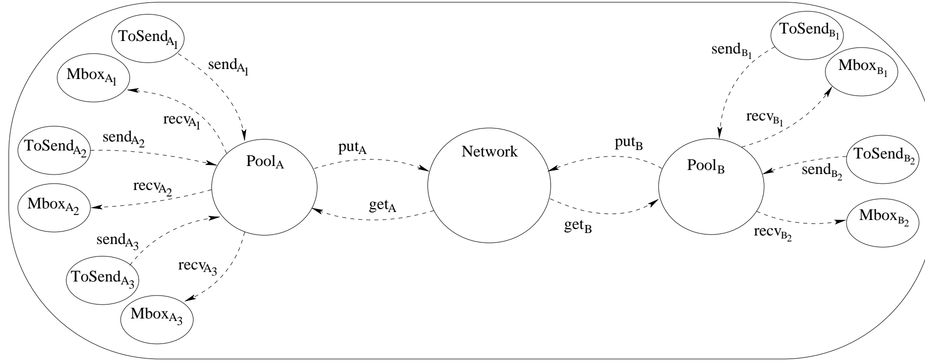

Fig. 5. Mail system.

$$
\begin{aligned}
\mathrm{send}_{d_i} = \ &\textbf{replace}\ \mathrm{ToSend}_{d_i} = \langle msg, \omega_t\rangle,\ \mathrm{Pool}_d = \langle \omega_p\rangle \\
&\textbf{by}\ \mathrm{ToSend}_{d_i} = \langle \omega_t\rangle,\ \mathrm{Pool}_d = \langle msg, \omega_p\rangle \\
\mathrm{recv}_{d_i} = \ &\textbf{replace}\ \mathrm{Pool}_d = \langle msg, \omega_p\rangle, \mathrm{Mbox}_{d_i} = \langle \omega_b\rangle \\
&\textbf{by}\ \mathrm{Pool}_d = \langle \omega_p\rangle, \mathrm{Mbox}_{d_i} = \langle msg, \omega_b\rangle \\
&\textbf{if}\ \mathit{recipient}(msg) = i
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{put}_d = \ &\textbf{replace}\ \mathrm{Pool}_d = \langle msg, \omega_p\rangle, \mathrm{Network} = \langle \omega_n\rangle \\
&\textbf{by}\ \mathrm{Pool}_d = \langle \omega_p\rangle, \mathrm{Network} = \langle msg, \omega_n\rangle \\
&\textbf{if}\ \mathit{recipientDomain}(msg) \neq d \\
\mathrm{get}_d = \ &\textbf{replace}\ \mathrm{Network} = \langle msg, \omega_n\rangle, \mathrm{Pool}_d = \langle \omega_p\rangle \\
&\textbf{by}\ \mathrm{Network} = \langle \omega_n\rangle, \mathrm{Pool}_d = \langle msg, \omega_p\rangle \\
&\textbf{if}\ \mathit{recipientDomain}(msg) = d
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{MailSystem} = \langle\ &\mathrm{send}_{A_1},\ \mathrm{recv}_{A_1},\ \mathrm{ToSend}_{A_1} = \langle\ldots\rangle,\ \mathrm{Mbox}_{A_1} = \langle\ldots\rangle, \\
&\mathrm{send}_{A_2},\ \mathrm{recv}_{A_2},\ \mathrm{ToSend}_{A_2} = \langle\ldots\rangle,\ \mathrm{Mbox}_{A_2} = \langle\ldots\rangle, \\
&\mathrm{send}_{A_3},\ \mathrm{recv}_{A_3},\ \mathrm{ToSend}_{A_3} = \langle\ldots\rangle,\ \mathrm{Mbox}_{A_3} = \langle\ldots\rangle, \\
&\mathrm{put}_A,\ \mathrm{get}_A,\ \mathrm{Pool}_A,\ \ \mathrm{Network},\ \ \mathrm{put}_B,\ \mathrm{get}_B,\ \mathrm{Pool}_B, \\
&\mathrm{send}_{B_1},\ \mathrm{recv}_{B_1},\ \mathrm{ToSend}_{B_1} = \langle\ldots\rangle,\ \mathrm{Mbox}_{B_1} = \langle\ldots\rangle, \\
&\mathrm{send}_{B_2},\ \mathrm{recv}_{B_2},\ \mathrm{ToSend}_{B_2} = \langle\ldots\rangle,\ \mathrm{Mbox}_{B_2} = \langle\ldots\rangle \\
&\rangle
\end{aligned}
$$

Fig. 6. Self-organization molecules.

– A server of a domain d is represented by two active molecules put_d and get_d.

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 recv_{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 5 represents graphically the solution with five clients grouped into two domains A and B and Figure 6 provides the definition of the 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 7). The emergency servers work with their own pool as usual but are active only

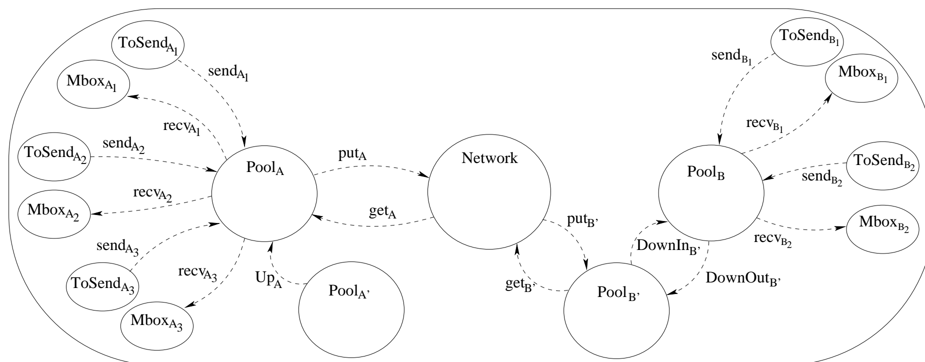

Fig. 7. Highly-available mail system.

when the corresponding main server has crashed. The modeling of a server crash can be done using the reactive molecules described in Figure 8. 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 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 Pool_d to $\text{Pool}_{d'}$; then, they may be forwarded by the emergency server.

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

$$\begin{aligned}
\text{crashServer}_d = &\ \textbf{replace}\ \text{put}_d,\ \text{get}_d,\ \text{Up}_d \\
&\ \textbf{by}\ \text{put}_{d'},\ \text{get}_{d'}, \text{DownIn}_d,\ \text{DownOut}_d \\
&\ \textbf{if}\ \mathit{failure}(d) \\
\text{repairServer}_d = &\ \textbf{replace}\ \text{put}_{d'},\ \text{get}_{d'}, \text{DownIn}_d,\ \text{DownOut}_d \\
&\ \textbf{by}\ \text{put}_d,\ \text{get}_d,\ \text{Up}_d \\
&\ \textbf{if}\ \mathit{recover}(d)
\end{aligned}$$

$$\begin{aligned}
\text{DownOut}_d = &\ \textbf{replace}\ \text{Pool}_d = \langle msg, \omega_p \rangle, \text{Pool}_{d'} = \langle \omega_n \rangle \\
&\ \textbf{by}\ \text{Pool}_d = \langle \omega_p \rangle, \text{Pool}_{d'} = \langle msg, \omega_n \rangle \\
&\ \textbf{if}\ \mathit{domain}(msg) \neq d \\
\text{DownIn}_d = &\ \textbf{replace}\ \text{Pool}_d = \langle \omega_p \rangle, \text{Pool}_{d'} = \langle msg, \omega_n \rangle \\
&\ \textbf{by}\ \text{Pool}_d = \langle msg, \omega_p \rangle, \text{Pool}_{d'} = \langle \omega_n \rangle \\
&\ \textbf{if}\ \mathit{domain}(msg) = d
\end{aligned}$$

$$\begin{aligned}
\text{Up}_d = &\ \textbf{replace}\ \text{Pool}_{d'} = \langle msg, \omega_p \rangle, \text{Pool}_d = \langle \omega_n \rangle \\
&\ \textbf{by}\ \text{Pool}_{d'} = \langle \omega_p \rangle, \text{Pool}_d = \langle msg, \omega_n \rangle
\end{aligned}$$

$$\begin{aligned}
\text{MailSystem} = \langle \ldots,\ \text{Up}_\text{A},\ \text{Up}_\text{B},\ \text{Pool}'_\text{A},\ \text{Pool}'_\text{B},\ &\text{crashServer}_\text{A},\ \text{repairServer}_\text{A}, \\
&\text{crashServer}_\text{B},\ \text{repairServer}_\text{B} \rangle
\end{aligned}$$

Fig. 8. Self-healing molecules.

DownIn_d transfers all messages bound to the domain d from the emergency pool $\text{Pool}_{d'}$ to the main pool 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 Up_d brings back all the messages of the emergency pool $\text{Pool}_{d'}$ into the the main pool 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 8 into the main solution.

3.3 Self-protection

Self-protection can be decomposed in two phases: a detection phase and a reaction phase. Detection consists in filtering data and reaction in preventing offensive data to spread (and sometimes also in counter-attacking). It can easily be expressed with the condition-reaction scheme of the chemical paradigm. In our mail system, self-protection is simply implemented with active molecules of the following form:

$$\text{self-protect} = \textbf{replace}\ x, \omega\ \textbf{by}\ \omega\ \textbf{if}\ \mathit{filter}(x)$$

If a molecule x is recognized as an offensive data by a filter function then it is suppressed. Variants of self-protect would consist in generating molecules to counter-attack or to send warnings.

Offensive data can take various forms such as spam, virus, ... A protection against spam can be represented by the molecule:

$$\text{rmSpam} = \textbf{replace } msg, \omega \textbf{ by } \omega \textbf{ if } isSpam(msg)$$

which is placed in a Pool_d solution. The contents of the pool can only be accessed when it is inert, that is when all spam messages have been suppressed by the active molecule rmSpam.

Two 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) 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 elegance of the resulting programs which rely essentially on the higher-order and chemical nature of Gamma. A direct implementation of our chemical specifications 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 the 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 γ-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 λ-calculus and a large class of non-deterministic programs [4]. However, in order to come close to a real chemical language, two fundamental extensions must be considered: reaction conditions and atomic capture. Along 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, reactive molecules (γ-abstractions) can be seen as catalysts that perform computations and implements new features. This programming style has been illustrated by the specification of an autonomic mail system in terms of solutions of molecules. Some molecules react 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. A distinctive and valuable property of our description is its modularity. Properties are described by independent collections of molecules and rules that are simply added to the system without requiring other changes.

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 research direction would be to pursue the extension of 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: Fifth International Workshop on Rule-Based Programming (RULE'04), Electronic Notes in Theoretical Computer Science (2004)
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: Proc. of the 13th Int. Conf. on Intelligent and Adaptive Systems and Software Engineering (IASSE'04). (2004)
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)