Chemical Specification of Autonomic Systems

Jean-Pierre Banâtre, Pascal Fradet, Yann Radenac

Chemical Specification of Autonomic Systems

J.-P. Banâtre, Y. Radenac

IRISA
Campus de Beaulieu
35042 Rennes Cedex, France
{jbanatre,yradenac}@irisa.fr

P. Fradet

INRIA Rhône-Alpes
655 avenue de l'Europe
38330 Montbonnot, France
pfradet@inria.fr

Abstract

Autonomic computing provides a vision of information systems allowing self-management of many predefined properties. Such systems take care of their own behavior and of their interactions with other components without any external intervention. One of the major challenges concerns the expression of properties and constraints of autonomic systems. We believe that the *chemical programming paradigm* (represented here by the Gamma formalism) is very relevant to the programming of autonomic systems. In Gamma, computation is described in terms of chemical reactions (rewrite rules) in solutions (multisets of elements). It captures the intuition of a collection of cooperative components which evolve freely according to some predefined constraints. In this paper, after a short presentation of a higher-order version of the Gamma formalism, it is shown through the example of a mailing system, how the major properties expected from an autonomic system can be easily expressed as a collection of chemical reactions.

1 Introduction

The Gamma formalism was proposed in [3] to capture the intuition of computation as the global evolution of a collection of atomic values interacting freely. Gamma can be introduced intuitively through the chemical reaction metaphor. The unique data structure in Gamma is the multiset which can be seen as a chemical solution. A simple program is made of a *reaction 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 more reactions can take place.

For example, the computation of the maximum element of a non empty set can be described as:

$$\textbf{replace } x, y \textbf{ by } x \textbf{ if } x > y$$

meaning that any couple of elements x and y of the multiset is replaced by x if the condition is fulfilled. This process goes on till a stable state is reached, that is to say, when only the maximum element remains. Note that, in this definition, nothing is said about the order of evaluation of the comparisons. If several disjoint pairs of elements satisfy the condition, the reactions can be performed in parallel.

The possibility of getting rid of artificial sequentiality in Gamma confers a high level nature to the language and allows the programmer to describe programs in a very abstract way. In some sense, one can say that it is possible in Gamma to express the very idea of an algorithm without any unnecessary linguistic idiosyncrasies. The interested reader may find in [3] a long series of examples (string processing problems, graph problems, geometry problems, ...) illustrating the Gamma style of programming and in [1] a review of contributions related to the chemical reaction model. Gamma has been applied to the development of operating systems and software architectures [4, 9, 10]; it has revealed many highly valuable properties as far as production of reliable software is concerned.

Autonomic computing provides a vision in which systems manage themselves according to some predefined goals. The essence of autonomic computing is self-organization. Like biological systems, "autonomic systems maintain and adjust their operation in the face of changing components, workloads, demands, and external conditions in the face of hardware or software failures, both innocent or malicious. The autonomic system might continually monitor its own use and check for component upgrades" [8]. We believe that the chemical programming paradigm (represented here by the Gamma formalism) is very relevant to the programming of autonomic systems. It captures the intuition of a collection of cooperative components which evolve freely according to some predefined constraints (reaction rules). System self-management arises as a result of interactions between members, in the same way as "intelligence" emerges from cooperation in colonies of biological

agents.

We describe the use of Gamma as a basis for the modeling and description of autonomic systems through the example of a mail system exhibiting many self-management properties. We develop our system by considering in turn, self-organization, self-healing, self-optimization, self-protection and self-configuration. Each property is described separately from the others by new rules which are simply added to the system description. This example shows that rule-based programming languages (and, in particular, chemical programming) allows the expression of such properties in a natural, elegant and modular way.

This article is organized as follows. Section 2 presents an overview of the γ-calculus [2], that summarizes in a simple and formal setting the main features of the chemical paradigm. Section 3 introduces higher-order Gamma, a high-level chemical programming language, and illustrates its style of programming through simple examples. Section 4 focuses on the application of higher-order Gamma to describe autonomic systems using a case study. Finally, section 5 discusses a few issues related to chemical autonomic systems and suggests avenues for further research.

2 The γ-calculus

In this section, we describe a higher-order calculus, the γ-calculus [2], that can be seen as a formal basis for the chemical paradigm (in much the same way as the λ-calculus is the formal basis of the functional paradigm). It generalizes the chemical model of computation by considering every element (including programs themselves) as a molecule. It naturally leads to a higher-order extension of the original Gamma language.

The fundamental data structure of the γ-calculus is the multiset (a collection which may contain several copies of the same element). Computation can be seen either intuitively, as chemical reactions between molecules moving freely in solutions, or formally, as associative, commutative and higher-order multiset rewritings. In the following, we summarize the very primitive features of the γ-calculus.

Molecules. Molecules (or γ-expressions) are variables, γ-abstractions, multisets or solutions of molecules. Their syntax is defined by the following grammar:

$$\begin{array}{rcll} M & ::= & x \mid y \mid \ldots & \textit{; variables} \\ & \mid & (\gamma\langle x\rangle.M) & \textit{; } \gamma\textit{-abstraction} \\ & \mid & (M_1, M_2) & \textit{; multiset} \\ & \mid & \langle M\rangle & \textit{; solution} \end{array}$$

Brownian motion. Brownian motion makes molecules move freely in a solution. In the γ-calculus, Brownian motion is represented by the associativity and commutativity of the multiset constructor ",". Equivalent molecules modulo associativity and commutativity (A/C rules) are denoted by "$\equiv$". For example:

$$(x,(y,x)) \equiv (y,(x,x)) \equiv y,x,x \equiv x,y,x$$

So, parentheses are not necessary to denote multisets. Solutions encapsulate molecules. Molecules can move within solutions but not across solutions. For example:

$$x,\langle y,x\rangle \equiv \langle x,y\rangle,\ x \not\equiv \langle x,x\rangle,y$$

Chemical reactions. Another distinctive feature of chemical models is the reaction concept. In the γ-calculus, it is represented by a rewrite rule called the γ-reduction:

$$(\gamma\langle x\rangle.M),\langle N\rangle \rightarrow_\gamma M[x := N] \quad \text{if } \langle N\rangle \text{ is inert} \quad \textit{; } \gamma\textit{-reduction}$$

If a γ-abstraction "meets" an inert solution, then they may react. The rule itself is similar to the β-reduction in the λ-calculus: the γ-abstraction $\gamma\langle x\rangle.M$ and the inert solution $\langle N\rangle$ are rewritten into the molecule M where all the free occurrences of the parameter x have been replaced by N. An inert solution is a solution where no reaction can occur. It is a solution that contains only abstractions or only solutions (however these inner solutions may not be inert). Note that γ-abstractions disappear in reactions: they are said to be *one-shot*.

The structural rules describe how reactions can occur in a γ-expression:

$$locality\ \frac{M_1 \rightarrow_\gamma M_2}{M, M_1 \rightarrow_\gamma M, M_2} \qquad solution\ \frac{M_1 \rightarrow_\gamma M_2}{\langle M_1\rangle \rightarrow_\gamma \langle M_2\rangle}$$

The $locality$ rule states that if a molecule M_1 can react then it can do so whatever its context M (i.e. the rest of the solution). The $solution$ rule states that reactions can occur within nested solutions.

Non-determinism. The A/C rules make the γ-calculus non-deterministic. If a molecule contains several elements, it is not know *a priori* how they will combine because of the Brownian motion. For example, consider the following (and very classical) example:

$$(\gamma\langle x\rangle.\gamma\langle y\rangle.x),\langle a\rangle,\langle b\rangle \rightarrow_\gamma \begin{cases} (\gamma\langle y\rangle.a),\langle b\rangle \rightarrow_\gamma a \\ (\gamma\langle y\rangle.b),\langle a\rangle \rightarrow_\gamma b \end{cases}$$

This molecule can reduce itself to two distinct stable terms depending on the application of A/C rules and whether the first reaction involves $\langle a\rangle$ or $\langle b\rangle$.

Higher-order. Finally, the γ-calculus is a higher-order model: abstractions are molecules and can be taken as parameter or yielded as result by other abstractions. This expressivity makes it possible to encode all standard programming language features (e.g. booleans, integers, pairs, recursion, ...) within the γ-calculus itself. For example, here is a possible encoding of the booleans and negation:

$$\begin{aligned}\mathbf{true} &\equiv \gamma\langle x\rangle.\gamma\langle y\rangle.x\\ \mathbf{false} &\equiv \gamma\langle x\rangle.\gamma\langle y\rangle.y\\ \mathbf{not} &\equiv \gamma\langle x\rangle.\langle\langle x\rangle, (\gamma\langle a\rangle.a, \langle\mathbf{false}\rangle)\rangle,\ (\gamma\langle b\rangle.b, \langle\mathbf{true}\rangle)\end{aligned}$$

The molecule $(\mathbf{not}, \langle\mathbf{true}\rangle)$ reacts as follows:

$$\begin{aligned}&(\gamma\langle x\rangle.\langle\langle x\rangle, (\gamma\langle a\rangle.a, \langle\mathbf{false}\rangle)\rangle, (\gamma\langle b\rangle.b, \langle\mathbf{true}\rangle)),\ \langle\mathbf{true}\rangle\\ &\xrightarrow{*}_\gamma\ \langle\langle\mathbf{true}\rangle, (\gamma\langle a\rangle.a, \langle\mathbf{false}\rangle)\rangle,\ (\gamma\langle b\rangle.b, \langle\mathbf{true}\rangle)\\ &\xrightarrow{*}_\gamma\ \langle\mathbf{true}, \langle\mathbf{false}\rangle\rangle,\ (\gamma\langle b\rangle.b, \langle\mathbf{true}\rangle)\\ &\equiv\ \langle\gamma\langle x\rangle.\gamma\langle y\rangle.x, \langle\mathbf{false}\rangle\rangle,\ (\gamma\langle b\rangle.b, \langle\mathbf{true}\rangle)\\ &\xrightarrow{*}_\gamma\ \langle\gamma\langle y\rangle.\,\mathbf{false}\rangle,\ (\gamma\langle b\rangle.b, \langle\mathbf{true}\rangle)\\ &\xrightarrow{*}_\gamma\ \mathbf{false}\end{aligned}$$

The γ-calculus is a minimal but very expressive model. It can encode the λ-calculus, but can also express non-deterministic primitives. However, as such, it is quite a clumsy programming tool. More practical programming languages can be built above the basic γ-calculus such as the reference Gamma language (used in this paper) presented in the next section.

3 A Higher-order Chemical Language

A practical chemical programming language can be built upon the γ-calculus by adding constants (e.g. booleans, integers), primitive operators (e.g. arithmetic operators, conditional), data structures (e.g. pairs, tuples), recursive definitions and pattern matching. We do not present this construction here but only introduce the constructions and notations used later on to express autonomic systems. The language that we use can be seen as a higher-order extension of Gamma [3]. Gamma has a very distinctive and elegant programming style which we illustrate by a collection of classical programs.

3.1 Higher-order Gamma

Here, 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 ($ident = M$). Furthermore, γ-abstractions (also called *active molecules*) can react according to a condition and can extract elements using pattern-matching. The syntax of γ-abstractions is extended to:

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

where M is the action, C is the condition of reaction and P a pattern extracting the elements participating in the reaction. Pattern 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, ...),
- ω 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 $m \equiv m_1, m_2$ 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 $\mathrm{Sol} = \langle x, y, \omega\rangle$ 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 ω.

The γ-abstractions are one-shot: they are consumed by the reaction. Many programs are naturally expressed by applying the same reaction an arbitrary number of times. We use recursive (or n-shot) γ-abstractions which are not consumed by the reaction. We denote them by the following syntax:

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

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, they can be removed by another molecule, thanks to the higher-order nature of the language. If the condition C is $\mathbf{true}$, we omit it in the definition of active (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).

3.2 Some classical examples

We now provide some simple examples to familiarize with Gamma's syntax and programming style.

Prime numbers. Given the function $x\ \mathrm{multipleOf}\ y$ which returns $\mathbf{true}$ if x is a multiple of y, the computation of the prime numbers up to n can be expressed by:

$$\langle \mathrm{prime}, 2, 3, 4, 5, \ldots, n\rangle$$

where the active molecule:

$$\text{prime} = \textbf{replace } x, y \textbf{ by } x \textbf{ if } y \text{ multipleOf } x$$

filters out all integers which can be divided by others. A stable state is reached only when all the remaining integers are prime. Let us point out the simplicity and elegance of this program compared to its expression in a traditional programming language.

The factorial function. The active molecule prod replaces two integers by their product.

$$\text{prod} = \textbf{replace } x, y \textbf{ by } x * y$$

This molecule can be used to compute the factorial of n:

$$\text{prod}, 2, 3, 4, 5, \ldots, n$$

Note that the classical way to write the factorial function involves a loop that computes the product in a specified order $(\ldots((2 * 3) * 4) * \ldots)$. Gamma does not have such an artificial sequential bias. Products are performed in any order.

In our example, the stable solution will be of the form $\langle \text{prod}, n! \rangle$. A function taking a solution $\langle 2, \ldots, n \rangle$ and returning only the integer $n!$ as result can be written as

$$\begin{aligned} & \text{fact} = \gamma\langle\omega\rangle.\text{clean}, \langle \text{prod}, \omega \rangle \\ \textit{with} \quad & \text{clean} = \gamma\langle \text{prod}, \omega \rangle.\omega \end{aligned}$$

The molecule fact takes a solution of integers, places prod inside whereas clean will remove it when the solution becomes stable.

4 Chemical Autonomic Systems

In this section, we develop programs which demonstrate how the chemical paradigm is adequate to describe autonomic systems. Examples are chosen in order to illustrate how the chemical paradigm facilitates the description of self-management properties.

4.1 Self-organization: a sorting machine

Consider the general problem of a system whose state must satisfy a number of properties but which is submitted to external and uncontrolled changes. This system must constantly re-organize itself to satisfy the properties. Let us illustrate this class of problem by a simple sorting example where the system state is made of pairs $index{:}value$ and the property of interest is that values are well-ordered (i.e. a smaller index means a smaller value). If the environment keeps adding random pairs in the state, the system must reorganize itself after each insertion of an ill-ordered element. In Gamma, the system is represented by

$$\text{State} = \langle \text{sort}, \ (i_1{:}v_1), \ \ldots, \ (i_n{:}v_n) \rangle$$

a solution made of pairs and of the following active molecule:

$$\begin{aligned} \text{sort} = \textbf{replace} \ & \ i{:}x, j{:}y \\ \textbf{by} \ & \ (j+1){:}x, j{:}y \\ \textbf{if} \ & \ i \leq j \text{ and } x > y \end{aligned}$$

The molecule sort looks for couples of ill-ordered values and increases the position of the greater values over the smaller ones. The solution evolves up to the point where no more reactions are possible: the solution has reached a stable state and the ordering property is satisfied. The "machine" is set to an initial environment

$$\text{Env} = \langle \text{alter}, \ \textit{seed}, \ \text{State} = \langle \text{sort} \rangle \rangle$$

where $seed$ is an integer used by alter to generate random integers. This molecule adds random integers at index 1 in the solution State:

$$\begin{aligned} \text{alter} = \textbf{replace} \ & \ \text{State} = \langle \omega \rangle, \ n \\ \textbf{by} \ & \ \text{State} = \langle 1{:}rand(n), \ \omega \rangle, \ n+1 \end{aligned}$$

New ill-ordered values break the "equilibrium" and violate the ordering property. However, sort searches continuously for new ill-ordered values so the state will reach a new stable state. When the state is stable, alter may add again a new value.

From that simple program, we can check properties. For example, if the program terminates, we can show that the pairs are well-ordered. The program terminates when no reaction can occur, so no couple of pairs satisfies the reaction condition: $\forall\ i{:}x, j{:}y\ (i > j) \vee (x \leq y)$ which is a definition for well-ordered values (i.e. a smaller index means a smaller value). Termination can be proved by finding a termination function like [7] does.

This very simple example shows how Gamma naturally expresses self-organizing systems. A Gamma program is made of a collection of rules (active molecules) which react until a stable state is reached. Theses rules remain and are applied (without any external intervention) as soon as the solution is unstable again. As we will see in the mail system example, this way of expressing self-organization is very practical and will be used intensively.

4.2 A mail system

We now describe an autonomic mail system within the Gamma framework. It consists in mail 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.

General description: self-organization. The mail system (see Figure 1) 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 $sender$, $recipient$, $senderDomain$, $recipientDomain$, $body$, 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 $\mathrm{Network}$ 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}.
- 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).

$$\begin{aligned}\mathrm{send}_{d_i} = \textbf{replace } & \mathrm{ToSend}_{d_i} = \langle msg, \omega_t \rangle,\ \mathrm{Pool}_d = \langle \omega_p \rangle \\ \textbf{by } & \mathrm{ToSend}_{d_i} = \langle \omega_t \rangle,\ \mathrm{Pool}_d = \langle msg, \omega_p \rangle\end{aligned}$$

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).

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

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.

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

The molecule get_d extracts messages addressed to d from the network and places them in the pool of domain d.

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

The system is a solution, named $\mathrm{MailSystem}$, containing molecules representing clients, messages, pools, servers, mailboxes and the network. Figure 1 represents graphically the following solution with five clients grouped into two domains A and B:

$$\begin{aligned}&\mathrm{MailSystem} = \langle \\ &\ \mathrm{send}_{\mathrm{A}_1},\ \mathrm{recv}_{\mathrm{A}_1},\ \mathrm{ToSend}_{\mathrm{A}_1} = \langle \ldots \rangle,\ \mathrm{Mbox}_{\mathrm{A}_1} = \langle \ldots \rangle, \\ &\ \mathrm{send}_{\mathrm{A}_2},\ \mathrm{recv}_{\mathrm{A}_2},\ \mathrm{ToSend}_{\mathrm{A}_2} = \langle \ldots \rangle,\ \mathrm{Mbox}_{\mathrm{A}_2} = \langle \ldots \rangle, \\ &\ \mathrm{send}_{\mathrm{A}_3},\ \mathrm{recv}_{\mathrm{A}_3},\ \mathrm{ToSend}_{\mathrm{A}_3} = \langle \ldots \rangle,\ \mathrm{Mbox}_{\mathrm{A}_3} = \langle \ldots \rangle, \\ &\ \mathrm{put}_{\mathrm{A}},\ \mathrm{get}_{\mathrm{A}},\ \mathrm{Pool}_{\mathrm{A}},\ \mathrm{Network},\ \mathrm{put}_{\mathrm{B}},\ \mathrm{get}_{\mathrm{B}},\ \mathrm{Pool}_{\mathrm{B}}, \\ &\ \mathrm{send}_{\mathrm{B}_1},\ \mathrm{recv}_{\mathrm{B}_1},\ \mathrm{ToSend}_{\mathrm{B}_1} = \langle \ldots \rangle,\ \mathrm{Mbox}_{\mathrm{B}_1} = \langle \ldots \rangle, \\ &\ \mathrm{send}_{\mathrm{B}_2},\ \mathrm{recv}_{\mathrm{B}_2},\ \mathrm{ToSend}_{\mathrm{B}_2} = \langle \ldots \rangle,\ \mathrm{Mbox}_{\mathrm{B}_2} = \langle \ldots \rangle \\ &\rangle\end{aligned}$$

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 2). The 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 following higher-order reaction:

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

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:

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

Figure 1. Mail system.

Figure 2. Highly-available mail system.

represents the recovery of the server.

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

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

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

The molecule DownIn_d transfers all messages bound to the domain d from the emergency pool $\mathrm{Pool}_{d'}$ to the main pool Pool_d.

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

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 emergency pool $\mathrm{Pool}_{d'}$ into the the main pool Pool_d to be then treated by the main server working again.

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

In our example, self-healing can be implemented by two emergency servers A' and B' and boils down to adding the following molecules to the solution:

$$\begin{aligned}\mathrm{MailSystem} = \langle \ldots, & \mathrm{Up_A}, \mathrm{Up_B}, \mathrm{Pool'_A}, \mathrm{Pool'_B}, \\ & \mathrm{crashServer_A}, \mathrm{repairServer_A}, \\ & \mathrm{crashServer_B}, \mathrm{repairServer_B} \rangle\end{aligned}$$

Self-optimization. The emergency server can also be used to treat messages even if the main server is up. This improves efficiency by allowing parallelization. We can activate the emergency server when the main server is up as follows:

$$\begin{aligned}\mathrm{optimize}_d = \textbf{replace } & \mathrm{Up}_d \\ \textbf{by } & \mathrm{put}_{d'}, \mathrm{get}_{d'}, \mathrm{balance}_d, \mathrm{balance}_{d'}\end{aligned}$$

The role of the two molecules $\mathrm{balance}_d$ and $\mathrm{balance}_{d'}$ is to perform dynamic load balancing between the two pools Pool_d and $\mathrm{Pool}_{d'}$.

$$\begin{aligned}\mathrm{balance}_d = \textbf{replace } & \mathrm{Pool}_d = \langle msg, \omega_p \rangle, \ \mathrm{Pool}_{d'} = \langle \omega_s \rangle \\ \textbf{by } & \mathrm{Pool}_d = \langle \omega_p \rangle, \ \mathrm{Pool}_{d'} = \langle msg, \omega_s \rangle \\ \textbf{if } & Card(\omega_p) > Card(\omega_s)\end{aligned}$$

$$\begin{aligned}\mathrm{balance}_{d'} = \textbf{replace } & \mathrm{Pool}_d = \langle \omega_p \rangle, \ \mathrm{Pool}_{d'} = \langle msg, \omega_s \rangle \\ \textbf{by } & \mathrm{Pool}_d = \langle msg, \omega_p \rangle, \ \mathrm{Pool}_{d'} = \langle \omega_s \rangle \\ \textbf{if } & Card(\omega_p) < Card(\omega_s)\end{aligned}$$

We have to model the crash of the main server when the system is running in the optimized mode.

$$\begin{aligned}\mathrm{crashOpt}_d = \textbf{replace } & \mathrm{balance}_d, \ \mathrm{balance}_{d'} \\ \textbf{by } & \mathrm{DownIn}_d, \ \mathrm{DownOut}_d \\ \textbf{if } & failure(d)\end{aligned}$$

The two molecules $\mathrm{balance}_d$ and $\mathrm{balance}_{d'}$ which characterize the optimized mode are removed and replaced by the molecules DownIn_d and $\mathrm{DownOut}_d$ which characterize the down state. After the main server is repaired the system may well trigger in the optimized mode again.

Adding, this self optimizing feature to our mail system boils down to adding the following molecules to the previous solution:

$$\begin{aligned}\mathrm{MailSystem} = \langle \ldots, & \mathrm{optimize_A}, \ \mathrm{optimize_B}, \\ & \mathrm{crashOpt_A}, \ \mathrm{crashOpt_B} \rangle\end{aligned}$$

Self-protection. Self-protection can be decomposed in two phases: a detection phase and a reaction phase. Detection consists mainly in filtering data (pattern matching). Reaction consists in preventing offensive data to spread and sometimes also in counter-attacking. This mechanism 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:

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

If a molecule x is recognized as an offensive data by a filter function then it is suppressed. Variants of $\mathrm{self\text{-}protect}$ 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:

$$\mathrm{rmSpam} = \mathbf{replace}\ msg\ \mathbf{by}\ \emptyset\ \mathbf{if}\ isSpam(msg)$$

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.

Self-configuration. We now consider adaptation and configuration issues that may arise with mobility. Assume that clients travel, move from personal computers to mobile phones, etc. Changes of environment suggest that clients should be able to migrate from a domain to another (closer or better suited to their new computing environment). We assume that the boolean $goTo_{d\text{-}e_i}$ signals to a client i in the domain d that it should go to the domain e. Such a migration can be described as follows:

$$\begin{aligned}\mathrm{migrate}_{d\text{-}e_i} = \mathbf{replace}\ & \mathrm{send}_{d_i},\ \mathrm{recv}_{d_i},\\ & \mathrm{Mbox}_{e_i},\ \mathrm{ToSend}_{e_i}\\ & \mathrm{Mbox}_{d_i},\ \mathrm{ToSend}_{d_i}\\ \mathbf{by}\ & \mathrm{send}_{e_i},\ \mathrm{recv}_{e_i},\\ & \mathrm{Mbox}_{d_i},\ \mathrm{ToSend}_{d_i},\ \langle\text{Fwd:i:d:e}\rangle\\ \mathbf{if}\ & goTo_{a\text{-}b_i}\end{aligned}$$

From now, the client will send and receive its messages from Pool_e. It may still have messages in its previous domain or some messages were sent before the migration and are in the network after the migration, or other clients may still send messages to its previous address. The migration places the tagged molecule $\langle$Fwd:i:d:e$\rangle$ in the solution in order to signal that messages to client i must be forwarded from domain (pool) d to domain e. The forward program between two domains d and e is:

$$\begin{aligned}&\mathrm{forward}_{d\text{-}e} =\\ &\mathbf{replace}\ \mathrm{Pool}_d = \langle msg, \omega_p\rangle,\ \langle\text{Fwd:i:d:e}\rangle\\ &\quad\mathbf{by}\ \mathrm{Pool}_d = \langle newMsg(msg, e_i), \omega_p\rangle\\ &\quad\mathbf{if}\ recipient(msg) = i\end{aligned}$$

where $newMsg(msg, r)$ builds a new message whose body is msg and recipient is r.

Notice that forwards accumulate when a client migrates several times (and remain even when the client is back to its original domain). We may prevent such forward chaining by using the following molecules:

$$\begin{aligned}\mathrm{shortcut} = \mathbf{replace}\ & \langle\text{Fwd:i:a:b}\rangle,\ \langle\text{Fwd:i:b:c}\rangle\\ \mathbf{by}\ & \langle\text{Fwd:i:a:c}\rangle,\ \langle\text{Fwd:i:b:c}\rangle\end{aligned}$$

$$\mathrm{shortcut}' = \mathbf{replace}\ \langle\text{Fwd:i:d:d}\rangle\ \ \mathbf{by}\ \emptyset$$

All these programs could (and should) be described generically (i.e. only once for all possible clients, source and destination domains). This would be easily done by tagging molecules (e.g. pools, clients, etc.). To simplify the presentation, we have described programs as if they were written for each client (server, domain, etc.). For the same reasons, we have also left tagging (e.g. of messages) implicit using extraction functions.

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. The self-healing, self-optimization and self-configuration programs are particularly illustrative of the elegance and power of the approach. A direct implementation of this program is likely to be quite inefficient and further refinements are needed; however, this is another exciting research direction, not tackled in the present paper.

5 Conclusion

In this paper, we have presented a higher-order extension of a multiset transformation language, called Gamma, which can be described using a chemical reaction metaphor. Let us emphasize that the higher-order property of our model makes it much more powerful and expressive than the original Gamma [3] or than the Linda language as described in [5]. We have studied the application of this formalism to the specification of autonomic systems through examples. Autonomic systems are described as molecules and transformation rules. These rules may apply as soon as a predefined condition holds, this happening 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 abstract description of autonomic systems. Reaction rules exhibit the essence of "autonomy" without going into useless details too early in the development process. A very 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.

Other authors have proposed to describe autonomous systems (such as robots) and plan their behavior in terms of chemical "machinery" [6]. Although the problem is quite different from the one considered here, the simplicity and elegance of the chemical metaphor is well illustrated.

We do not pretend that our approach solves all problems in a straightforward manner. We simply believe that this unconventional model of programming is well-suited to abstract and modular descriptions of self-managing (autonomic) systems. It may well be a source of inspiration for

the design of new specification languages dedicated to autonomic systems. Formal studies of Gamma programs can be carried out to prove properties on them like [4] does. For example, "not losing any messages" can be an interesting property to prove for our mail system. Finally, the language could be improved to avoid the clumsy encodings (e.g. by adding data structures) and other high level facilities.

References

[1] J.-P. Banâtre, P. Fradet, and D. Le Métayer. Gamma and the chemical reaction model: Fifteen years after. In *Multiset Processing*, volume 2235 of *LNCS*, pages 17–44. Springer-Verlag, 2001.

[2] J.-P. Banâtre, P. Fradet, and Y. Radenac. Principles of chemical programming. In *RULE'04*, ENTCS. Elsevier, 2004. To appear ...

[3] J.-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *Communications of the ACM (CACM)*, 36(1):98–111, Jan. 1993.

[4] H. Barradas. *Systematic derivation of an operating system kernel in Gamma*. Thesis (in french), University of Rennes 1, France, July 1993.

[5] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.

[6] A. D'Angelo. Using a chemical metaphor to implement autonomous systems. In *Topics in Artificial Intelligence*, volume 992 of *LNAI*, pages 315–322, Florence, Oct. 1995.

[7] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, Aug. 1979.

[8] J. Kephart and D. Chess. The vision of autonomic computing. *IEEE Computer*, Jan. 2003.

[9] D. Le Métayer. Software architecture styles as graph grammars. In *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 15–23. ACM Press, 1996.

[10] D. Mentré, D. Le Métayer, and T. Priol. Formalization and verification of coherence protocols with the Gamma framework. In *Proceedings of the 5th International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE-2000)*. ACM, June 2000.

[11] D. Patterson, A. Brown, P. Broadwell, et al. Recovery oriented computing (ROC): Motivations, definition, techniques and case studies. Technical Report CSD-02-1175, University of California at Berkeley, Mar. 2002.