# Programming Self-Organizing Systems with the Higher-Order Chemical Language

JEAN-PIERRE BANÂTRE[1], PASCAL FRADET[2], YANN RADENAC[1]

[1] *INRIA / IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France*
[2] *INRIA Rhône-Alpes, 655 avenue de l'Europe, 38330 Montbonnot, France*

jbanatre@irisa.fr, Pascal.Fradet@inria.fr, yradenac@irisa.fr

Received on 17th March 2006

In a chemical language, computation is viewed as abstract molecules reacting in an abstract chemical solution. Data can be seen as molecules and operations as chemical reactions: if some molecules satisfy a reaction condition, they are replaced by the result of the reaction. When no reaction is possible within the solution, a normal form is reached and the program terminates. In this article, we introduce HOCL, the Higher-Order Chemical Language, where reaction rules are also considered as molecules. We illustrate the application of HOCL to the specification of self-organizing systems. We describe two case studies: an autonomic mail system and the coordination of an image-processing pipeline on a grid.

*Key words:* Chemical programming, higher-order multiset rewriting, self-organization, autonomic systems, coordination

## 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 (data-structure that allows several occurrences of the same element). Computation proceeds by rewritings which consume and produce new elements according to 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 [5] and later extended in [6]. 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 (or inert) state is reached, that is to say, when no reaction can take place anymore. For example, the reaction

$$\mathrm{max} = \mathbf{replace}\, x, y\, \mathbf{by}\, x\, \mathbf{if}\, x \geq y$$

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

$$\mathrm{primes} = \mathbf{replace}\, x, y\, \mathbf{by}\, y\, \mathbf{if}\, x \;\mathrm{div}\; y$$

computes the prime numbers lower or equal to a given number $N$ when applied to the multiset of all numbers between 2 and $N$ ($x \;\mathrm{div}\; y$ is true if and only if $x$ divides $y$).

Let us emphasize the conciseness and elegance of Gamma programs. Programs can be expressed without artificial sequentiality (i. e., unrelated to the logic of the program). If several disjoint tuples of elements satisfy the condition, the reactions can be performed in parallel. The interested reader may find in [6] a long series of examples (string processing problems, graph problems, geometry problems, etc.) illustrating the Gamma programming style.

Let us point out that we consider the chemical reaction only as a metaphor allowing a fresh look at programs and computation. In the same spirit, the chemical reaction model has been used as a source of inspiration for many different works [2]. However, there are models, like artificial chemistries [8], which are closer to real chemical systems and, sometimes, even use real chemistry to compute. In contrast, our research focuses on the design of expressive and unconventional programming languages.

The Higher-Order Chemical Language (HOCL) is a higher-order extension of Gamma: chemical programs (reactions) are first-class citizens. HOCL increases the expressive power of Gamma by allowing one to write reactions that consume or produce other reactions. This feature greatly facilitates the expression of self-organizing systems. The main objective of this article is to illustrate these benefits through case studies. We first introduce HOCL informally using simple examples. Then, we present its use to the specification of two self-organizing systems: an autonomic mail system and the coordination of an image-processing pipeline on a grid.

## 2 THE HIGHER-ORDER CHEMICAL LANGUAGE

HOCL [4] is a higher-order extension of Gamma based on the $\gamma$-calculus [3]. Here, we present briefly and informally the features of HOCL used in the case studies of Sections 3 and 4. The interested reader will find a much more complete and formal presentation in [4].

In HOCL, programs, solutions, data and reactions are molecules. A program is a solution of atoms

$$\langle A_1, \ldots, A_n \rangle$$

that is, a multiset of atoms built using the associative and commutative operator ",". Associativity and commutativity formalize the Brownian motion of a chemical solution. They can always be used to reorganize molecules. Atoms are either basic constants (integers, booleans, etc.), sub-solutions ($\langle M \rangle$), pairs ($A_1{:}A_2$) or reaction rules. A reaction rule is written

$$\textbf{replace-one}\, P\, \textbf{by}\, M\, \textbf{if}\, C$$

where $P$ is a pattern which matches the required atoms, $C$ is the reaction condition and $M$ the result of the reaction. For example,

$$\langle (\textbf{replace-one}\, x{::}Int\, \textbf{by}\, 9\, \textbf{if}\, x \geq 10), 4, 9, 15 \rangle \rightarrow \langle 4, 9, 9 \rangle$$

That reaction rule resembles a ceiling function. Its pattern $x{::}Int$ matches an integer, the condition imposes the integer to be greater than $10$ and the action replaces it by $9$. In the rest of this article, we omit types in patterns when there is no ambiguity. For example, it is clear that the previous pattern must select an integer since the condition is $x \geq 10$; the previous reaction can be written $\textbf{replace-one}\, x\, \textbf{by}\, 9\, \textbf{if}\, x \geq 10$ instead.

Such reaction rules are said to be *one-shot* since they are consumed when they react. Reactions rules which remain after they react are called $n$*-shot*. Like in Gamma, there are denoted by $\textbf{replace}\, P\, \textbf{by}\, M\, \textbf{if}\, C$. The execution of a chemical program consists in performing reactions (non deterministically and possibly in parallel) until the solution becomes inert i. e., no reaction can take place anymore. For example, the following program computes the prime numbers lower than 10 using a chemical version of the Eratosthenes' sieve:

$$\langle(\mathbf{replace}\, x, y\, \mathbf{by}\, x\, \mathbf{if}\, x \text{ div } y), 2, 3, 4, 5, 6, 7, 8, 9, 10\rangle$$

The reaction removes any element $y$ which can be divided by another one $x$. Initially several reactions are possible. For example, the couple $(2, 10)$ can be replaced by 2, the couple $(3, 9)$ by 3 or $(4, 8)$ by $4$ etc. The solution becomes inert when the rule cannot react with any couple of integers in the solution, that is to say, when the solution contains only prime numbers. The result of the computation in our example is $\langle(\mathbf{replace}\, x, y\, \mathbf{by}\, x\, \mathbf{if}\, x \text{ div } y), 2, 3, 5, 7\rangle$.

A molecule inside a solution cannot react with a molecule outside the solution (the construct $\langle . \rangle$ can be seen as a membrane). Reaction rules can access the contents of a sub-solution only if it is inert. This important restriction introduces some sequentiality in an otherwise highly parallel model: all reactions should be performed in a sub-solution before its content may be accessed or extracted. So, the pattern $\langle P\rangle$ matches only inert solutions whose (inert) content matches the pattern $P$.

Reactions can be named (or tagged) using the syntax $\text{name} = \mathbf{replace} \ldots$. Names are used to match and extract specific reactions using the same syntax ($\text{name} = x$). We often use the **let** operator to name reactions and assume that

$$\mathbf{let}\, \text{name} = M\, \mathbf{in}\, N \quad \stackrel{\text{def}}{=} \quad N[(\text{name} = M)/\text{name}]$$

that is, the occurrences of $\text{name}$ in $N$ are replaced by $\text{name} = M$.

We also often make use of the pattern $\omega$ which can match any molecule even the "empty one". This pattern is very convenient to extract elements from a solution.

Using all these features, the previous example can be rewritten in order to remove the reaction at the end of the computation:

$$\begin{array}{l}
\mathbf{let}\ \text{sieve} = \mathbf{replace}\, x, y\, \mathbf{by}\, x\, \mathbf{if}\, x \text{ div } y\, \mathbf{in} \\
\mathbf{let}\ \text{clean} = \mathbf{replace\text{-}one} \langle \text{sieve} = x, \omega\rangle\, \mathbf{by}\, \omega\, \mathbf{in} \\
\qquad \langle \text{clean}, \langle \text{sieve}, 2, 3, 4, 5, 6, 7, 8, 9, 10\rangle\rangle
\end{array}$$

The reduction proceeds as follows:

$$\begin{array}{ll}
\multicolumn{2}{l}{\langle \text{clean} = \ldots, \langle \text{sieve} = \ldots, 2, 3, 4, 5, 6, 7, 8, 9, 10\rangle\rangle} \\
\xrightarrow{*} & \langle \text{clean} = \ldots, \langle \text{sieve} = \ldots, 2, 3, 5, 7\rangle\rangle \\
\rightarrow & \langle 2, 3, 5, 7\rangle
\end{array}$$

The reaction rule $\mathrm{clean}$ cannot be applied until the sub-solution is inert. Only $\mathrm{sieve}$ can react until all primes are computed. Then, the one-shot rule $\mathrm{clean}$ extracts the prime numbers and suppresses the reaction rule $\mathrm{sieve}$.

## 3 AUTONOMIC SYSTEMS IN HOCL

New advances in networking and computing technology has produced an explosive growth in networked applications and information services. These applications are more and more complex, heterogeneous and dynamic by essence. This combination of new parameters results in application development, configuration and management which break present computing paradigms. These applications should be able to manage themselves and react without external intervention. The goal of Autonomic Computing is to realize computer and software systems and applications that can manage themselves in accordance with high-level guidance from humans [11, 12].

The Chemical paradigm is well suited to express autonomic properties. Consider the general problem of a system whose state must satisfy a number of invariant properties but which is submitted to external and uncontrolled changes. This system must constantly re-organize itself to satisfy the properties. Typically in a chemical setting, an invariant property is represented by a $n$-shot reaction rule aiming at reaching a stable state where the property is satisfied. An autonomic program is made of a collection of such rules which are applied (without any external intervention) as soon as the solution is unstable again. This way of expressing self-management is very practical and will be used intensively.

To support our claims, we now specify an autonomic mail system within the higher-order chemical framework. The system consists of 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 by a solution that uses 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 $sender$, $recipient$, $body$, $senderDomain$, $recipientDomain$, etc.).

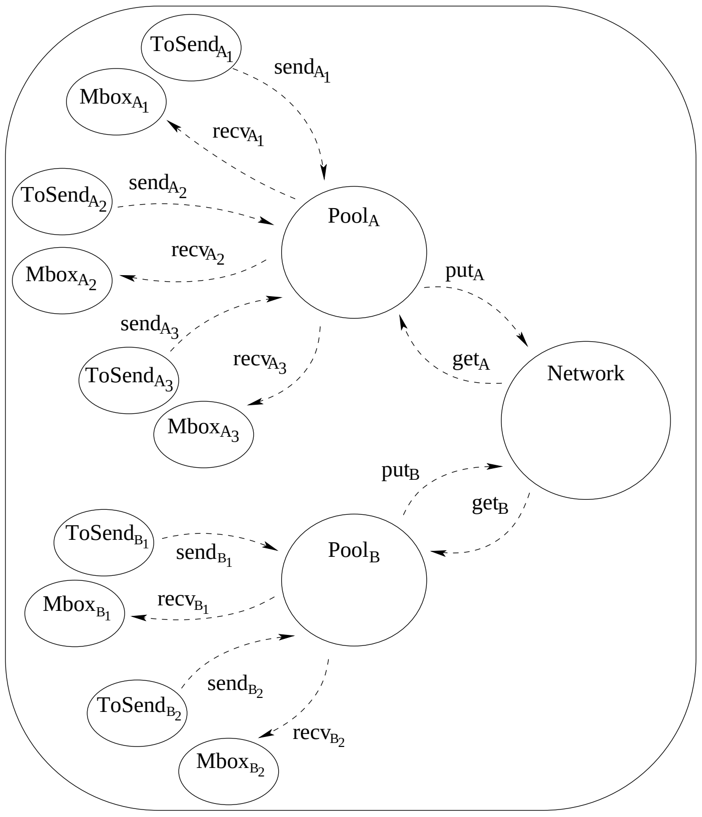

Figure 1
An autonomic mail system as a chemical solution.

- Solutions named $\mathrm{ToSend}_{d_i}$ contain the messages to be sent by the client $i$ of domain $d$.
- Solutions named $\mathrm{Mbox}_{d_i}$ contain the messages received by the client $i$ of domain $d$.
- Solutions named $\mathrm{Pool}_d$ 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 $\mathrm{send}_{d_i}$ and $\mathrm{recv}_{d_i}$.
- A server of a domain $d$ is represented by two active molecules $\mathrm{put}_d$ and $\mathrm{get}_d$.

We do not specify within the system how new messages are produced by users. In any case, the mail system has no problem to deal with $\mathrm{ToSend}$ solutions changing dynamically. 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.

Messages stores are represented by sub-solutions. Movement of messages are made by reaction rules of the form:

$$\begin{array}{r l}\textbf{replace} & A{:}\langle \mathit{msg}, \omega_A\rangle,\ B{:}\langle \omega_B\rangle \\ \textbf{by} & A{:}\langle \omega_A\rangle,\ B{:}\langle \mathit{msg}, \omega_B\rangle \\ \textbf{if} & \mathit{Condition}\end{array}$$

which moves the message $\mathit{msg}$ from the store $A$ to the store $B$ if $\mathit{Condition}$ is satisfied.

Figure 2 gives the molecules that route messages in a self-organizing way. The $\mathrm{send}_{d_i}$ molecule sends messages of the client $i$ (i. e., messages in the $\mathrm{ToSend}_{d_i}$ solution) to the client's domain pool (i. e., the $\mathrm{Pool}_d$ solution). The $\mathrm{recv}_{d_i}$ molecule places the messages addressed to client $i$ (i. e., messages in the $\mathrm{Pool}_d$ solution whose recipient is $i$) in the client's mailbox (i. e., the $\mathrm{Mbox}_{d_i}$ solution). Servers forward messages from their pool to the network. They receive messages from the network and store them in their pool. The $\mathrm{put}_d$ molecule forwards only messages addressed to other domains than $d$. The molecule $\mathrm{get}_d$ extracts messages addressed to domain $d$ from the network and places them in the pool of domain $d$. The system is a solution, named $\mathrm{MailSystem}$, containing molecules representing clients, messages, pools, servers, mailboxes and the network. Figure 1 represents graphically the system with five clients grouped into two domains $A$ and $B$.

**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 3 and 4). 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 higher-order reaction rule $\mathrm{crashServer}_d$: the active molecules representing a main server are replaced by molecules representing the corresponding emergency server. The

$$
\begin{aligned}
\mathrm{send}_{d_i} &= \textbf{replace}\ \mathrm{ToSend}_{d_i}{:}\langle msg, \omega_t\rangle,\ \mathrm{Pool}_d{:}\langle \omega_p\rangle \\
&\qquad \textbf{by}\ \mathrm{ToSend}_{d_i}{:}\langle \omega_t\rangle,\ \mathrm{Pool}_d{:}\langle msg, \omega_p\rangle \\
\mathrm{recv}_{d_i} &= \textbf{replace}\ \mathrm{Pool}_d{:}\langle msg, \omega_p\rangle, \mathrm{Mbox}_{d_i}{:}\langle \omega_b\rangle \\
&\qquad \textbf{by}\ \mathrm{Pool}_d{:}\langle \omega_p\rangle, \mathrm{Mbox}_{d_i}{:}\langle msg, \omega_b\rangle \\
&\qquad \textbf{if}\ \mathit{recipient}(msg) = i \\
\mathrm{put}_d &= \textbf{replace}\ \mathrm{Pool}_d{:}\langle msg, \omega_p\rangle, \mathrm{Network}{:}\langle \omega_n\rangle \\
&\qquad \textbf{by}\ \mathrm{Pool}_d{:}\langle \omega_p\rangle, \mathrm{Network}{:}\langle msg, \omega_n\rangle \\
&\qquad \textbf{if}\ \mathit{recipientDomain}(msg) \neq d \\
\mathrm{get}_d &= \textbf{replace}\ \mathrm{Network}{:}\langle msg, \omega_n\rangle, \mathrm{Pool}_d{:}\langle \omega_p\rangle \\
&\qquad \textbf{by}\ \mathrm{Network}{:}\langle \omega_n\rangle, \mathrm{Pool}_d{:}\langle msg, \omega_p\rangle \\
&\qquad \textbf{if}\ \mathit{recipientDomain}(msg) = d
\end{aligned}
$$

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

Figure 2
Self-organizing routing molecules and the main solution.

boolean *failure* denotes a (potentially complex) failure detection mechanism. The inverse reaction $\mathrm{repairServer}_d$ represents the recovery of the server.

The two molecules $\mathrm{Up}_d$ 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 $\mathrm{Pool}_d$ 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 $\mathrm{Pool}_d$ to the emergency pool $\mathrm{Pool}_{d'}$. The molecule $\mathrm{DownIn}_d$ transfers all messages bound to the domain $d$ from the emergency pool $\mathrm{Pool}_{d'}$ to the main pool $\mathrm{Pool}_d$. After a transition from the down state to the up state, it may remain some messages in the emergency pools. So, the molecule $\mathrm{Up}_d$ brings back all the messages of emergency pool $\mathrm{Pool}_{d'}$ into

$$
\begin{aligned}
\mathrm{crashServer}_d = {} & \textbf{replace}\ \mathrm{put}_d,\ \mathrm{get}_d,\ \mathrm{Up}_d \\
& \textbf{by}\ \mathrm{put}_{d'},\ \mathrm{get}_{d'}, \mathrm{DownIn}_d,\ \mathrm{DownOut}_d \\
& \ \textbf{if}\ \mathit{failure}(d) \\
\mathrm{repairServer}_d = {} & \textbf{replace}\ \mathrm{put}_{d'},\ \mathrm{get}_{d'}, \mathrm{DownIn}_d,\ \mathrm{DownOut}_d \\
& \textbf{by}\ \mathrm{put}_d,\ \mathrm{get}_d,\ \mathrm{Up}_d \\
& \ \textbf{if}\ \mathit{recover}(d) \\
\mathrm{DownOut}_d = {} & \textbf{replace}\ \mathrm{Pool}_d{:}\langle \mathit{msg}, \omega_p\rangle, \mathrm{Pool}_{d'}{:}\langle \omega_n\rangle \\
& \textbf{by}\ \mathrm{Pool}_d{:}\langle \omega_p\rangle, \mathrm{Pool}_{d'}{:}\langle \mathit{msg}, \omega_n\rangle \\
& \ \textbf{if}\ \mathit{domain}(\mathit{msg}) \neq d \\
\mathrm{DownIn}_d = {} & \textbf{replace}\ \mathrm{Pool}_d{:}\langle \omega_p\rangle, \mathrm{Pool}_{d'}{:}\langle \mathit{msg}, \omega_n\rangle \\
& \textbf{by}\ \mathrm{Pool}_d{:}\langle \mathit{msg}, \omega_p\rangle, \mathrm{Pool}_{d'}{:}\langle \omega_n\rangle \\
& \ \textbf{if}\ \mathit{domain}(\mathit{msg}) = d \\
\mathrm{Up}_d = {} & \textbf{replace}\ \mathrm{Pool}_{d'}{:}\langle \mathit{msg}, \omega_p\rangle, \mathrm{Pool}_d{:}\langle \omega_n\rangle \\
& \textbf{by}\ \mathrm{Pool}_{d'}{:}\langle \omega_p\rangle, \mathrm{Pool}_d{:}\langle \mathit{msg}, \omega_n\rangle
\end{aligned}
$$

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

Figure 3
Crash and self-healing molecules.

the main pool $\mathrm{Pool}_d$ to be then treated by the main server working again. In our example, self-healing can be implemented by two emergency servers $A'$ and $B'$ and boils down to adding the corresponding self-healing molecules to the solution.

**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 with the molecules of Figure 5.

The role of the two molecules $\mathrm{balance}_d$ and $\mathrm{balance}_{d'}$ is to perform dynamic load balancing between the two pools $\mathrm{Pool}_d$ and $\mathrm{Pool}_{d'}$. Pools are balanced according to their number of messages (the function $Card$ returns the cardinal of a molecule, i. e., the number of messages). We have to model the crash of the main server when the system is running in the optimized

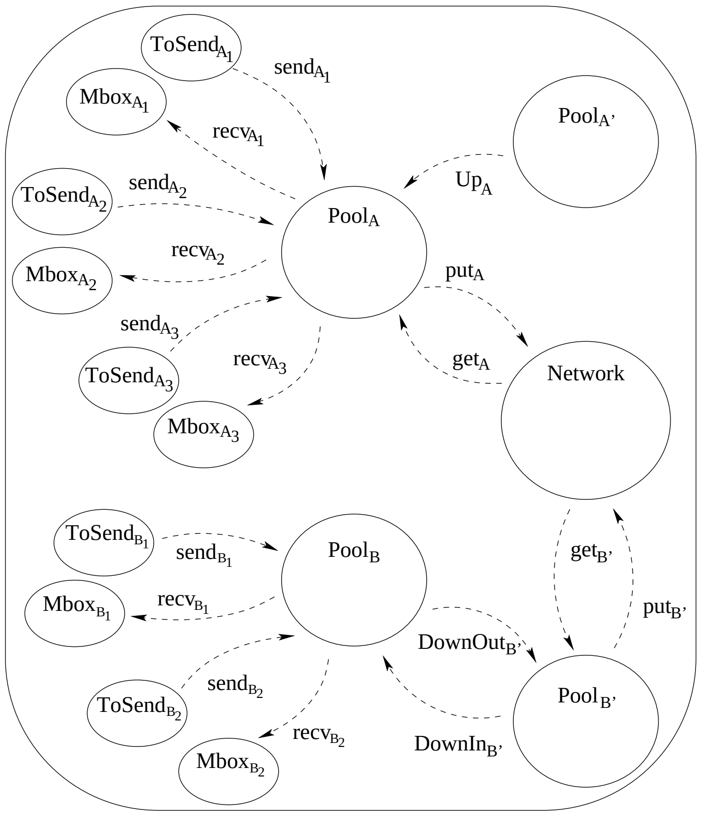

Figure 4
Highly-available mail system.

mode. The two molecules $\mathrm{balance}_d$ and $\mathrm{balance}_{d'}$ which characterize the optimized mode are removed and replaced by the molecules $\mathrm{DownIn}_d$ 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 these molecules to the main solution.

**Self-protection.** Self-protection can be decomposed in two phases: a detection phase and a reaction phase. The detection phase consists mainly in

$$\begin{aligned}
\mathrm{optimize}_d &= \textbf{replace } \mathrm{Up}_d \\
&\qquad \textbf{by } \mathrm{put}_{d'},\ \mathrm{get}_{d'},\ \mathrm{balance}_d,\ \mathrm{balance}_{d'} \\
\mathrm{balance}_d &= \textbf{replace } \mathrm{Pool}_d{:}\langle msg, \omega_p\rangle,\ \mathrm{Pool}_{d'}{:}\langle\omega_s\rangle \\
&\qquad \textbf{by } \mathrm{Pool}_d{:}\langle\omega_p\rangle,\ \mathrm{Pool}_{d'}{:}\langle msg, \omega_s\rangle \\
&\qquad\ \textbf{if } \mathit{Card}(\omega_p) > \mathit{Card}(\omega_s) \\
\mathrm{balance}_{d'} &= \textbf{replace } \mathrm{Pool}_d = \langle\omega_p\rangle,\ \mathrm{Pool}_{d'}{:}\langle msg, \omega_s\rangle \\
&\qquad \textbf{by } \mathrm{Pool}_d{:}\langle msg, \omega_p\rangle,\ \mathrm{Pool}_{d'}{:}\langle\omega_s\rangle \\
&\qquad\ \textbf{if } \mathit{Card}(\omega_p) < \mathit{Card}(\omega_s) \\
\mathrm{crashOpt}_d &= \textbf{replace } \mathrm{balance}_d,\ \mathrm{balance}_{d'} \\
&\qquad \textbf{by } \mathrm{DownIn}_d,\ \mathrm{DownOut}_d \\
&\qquad\ \textbf{if } \mathit{failure}(d)
\end{aligned}$$

$$\begin{aligned}
\mathrm{MailSystem}{:}\langle \ldots,\ &\mathrm{optimize}_{\mathrm{A}},\ \mathrm{optimize}_{\mathrm{B}}, \\
&\mathrm{crashOpt}_{\mathrm{A}},\ \mathrm{crashOpt}_{\mathrm{B}}\rangle
\end{aligned}$$

Figure 5
Self-optimizing molecules.

filtering data (pattern matching). The reaction phase consists in preventing offensive data from spreading 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:

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

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, etc. A protection against spam can be represented by the molecule:

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

which is placed in a $\mathrm{Pool}_d$ 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 by the rule $\mathrm{migrate}_{d\text{-}e_i}$ (see Figure 6).

$$\begin{aligned}
\mathrm{migrate}_{d\text{-}e_i} = \mathbf{replace}\ & \mathrm{send}_{d_i},\ \mathrm{recv}_{d_i}, \mathrm{Mbox}_{e_i},\ \mathrm{ToSend}_{e_i}, \mathrm{Mbox}_{d_i},\ \mathrm{ToSend}_{d_i} \\
\mathbf{by}\ & \mathrm{send}_{e_i},\ \mathrm{recv}_{e_i}, \mathrm{Mbox}_{d_i},\ \mathrm{ToSend}_{d_i},\ \langle \mathrm{Fwd}{:}i{:}d{:}e\rangle \\
\mathbf{if}\ & goTo_{a\text{-}b_i} \\
\mathrm{forward}_{d\text{-}e} = \mathbf{replace}\ & \mathrm{Pool}_d{:}\langle msg, \omega_p\rangle,\ \langle \mathrm{Fwd}{:}i{:}d{:}e\rangle \\
\mathbf{by}\ & \mathrm{Pool}_d{:}\langle newMsg(msg, e_i), \omega_p\rangle,\ \langle \mathrm{Fwd}{:}i{:}d{:}e\rangle \\
\mathbf{if}\ & recipient(msg) = i \\
\mathrm{shortcut} = \mathbf{replace}\ & \langle \mathrm{Fwd}{:}i{:}a{:}b\rangle,\ \langle \mathrm{Fwd}{:}i{:}b{:}c\rangle \\
\mathbf{by}\ & \langle \mathrm{Fwd}{:}i{:}a{:}c\rangle,\ \langle \mathrm{Fwd}{:}i{:}b{:}c\rangle \\
\mathrm{shortcut}' = \mathbf{replace}\ & \langle \mathrm{Fwd}{:}i{:}d{:}d\rangle, \omega\ \mathbf{by}\ \omega
\end{aligned}$$

Figure 6
Self-configuration molecules for migrations of users.

From now, the client will send and receive its messages from $\mathrm{Pool}_e$. 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 $\langle \mathrm{Fwd}{:}i{:}d{:}e\rangle$ in the solution in order to signal that messages to client $i$ must be forwarded from domain $d$ to domain $e$. The reaction rule $\mathrm{forward}_{d\text{-}e}$ forwards messages between two domains $d$ and $e$, where $newMsg(msg, r)$ builds a new message whose body is $msg$ and recipient is $r$.

Notice that forward molecules 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 molecules $\mathrm{shortcut}$ and $\mathrm{shortcut}'$.

**Remarks.** 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. Reaction rules exhibit the essence of "autonomy" without going into useless details too early in the development process. Even if we have not designed a methodology for reasoning on higher-order chemical programs yet, basic reasoning can rely on the semantics and equivalence laws of HOCL defined in [4]. The interested reader may refer to [10] which presents a calculus of Gamma programs and its application to program reasoning and refinement.

Let us emphasize the conciseness and modularity of the resulting programs which rely essentially on the higher-order and chemical nature of HOCL. The self-healing, self-optimization and self-configuration programs are particularly illustrative of the elegance and power of the approach. They are described by independent collections of rules that are simply added to the system without requiring other changes. Note however that a direct implementation of this system is likely to be quite inefficient and further refinements are needed. This is another exciting research direction, not tackled here.

## 4 COORDINATION WITH HOCL

This section presents another example of a self-organizing system. The objective is to execute a program on several resources, more precisely, on a computational grid [9]. On each resource runs a process (or agent) that achieves a part of the computation of the program and coordinates with other processes (or agents) on other resources.

**The basic application.** Our application is an image-processing pipeline (IPP) composed of two filters $F_1$ and $F_2$. For example, $F_1$ might reduce noise and a $F_2$ might increase contrast. The property to enforce is that filter $F_1$ must be applied before filter $F_2$ on each image. To ensure this, images are typed and filters are applied according to the type of images. Raw images have type $\mathrm{A}$, images returned by $\mathrm{F}_1$ have type $\mathrm{B}$ and images returned by $\mathrm{F}_2$ have type $\mathrm{C}$. The HOCL program implementing that pipeline is:

$$\begin{array}{l}\textbf{let}\,\mathrm{runF}_1 = \textbf{replace}\ x{::}\mathrm{A}\ \textbf{by}\ (\mathrm{F}_1\ x)\ \textbf{in}\\ \textbf{let}\,\mathrm{runF}_2 = \textbf{replace}\ x{::}\mathrm{B}\ \textbf{by}\ (\mathrm{F}_2\ x)\ \textbf{in}\\ \langle \mathrm{runF}_1, \mathrm{runF}_2, \mathrm{i}_1, \mathrm{i}_2, \ldots\rangle\end{array}$$

Initially, the solution has two reactions ($\mathrm{runF}_1$ and $\mathrm{runF}_2$) applying the filters and an arbitrary number of raw images $\mathrm{i}_1, \mathrm{i}_2, \ldots$ of type $\mathrm{A}$.

**Basic coordination on a grid.** A grid is a network of resources which in principle, can be used transparently like a single machine to execute programs, store data, etc. A grid is a dynamic system: resources become overloaded, new ones become available, etc. So, before launching a program on a grid, it is in general impossible to know what resources will be allocated to the program. Resource allocation is determined dynamically according to their characteristics: CPU type, load state, computing power, memory size, connectivity, communication cost, etc.

A major challenge is to describe the coordination of the execution of programs on a grid in order to ensure some qualities of services (QoS) like efficiency, security, fault-tolerance, etc.

The chemical paradigm is well suited to the coordination of programs on a grid. Like in a chemical program where reactions occur in a parallel and chaotic way, communications (or coordinations) occur in a parallel and chaotic way between resources. Coordination between resources can be expressed by chemical reactions. Reactions rules specify different coordination that may occur in a solution of resources. For example, the system programmer may specify the dynamic placement or the migration of tasks using reaction rules. A load-balancing rule may specify that a task allocated to a resource should migrate to a less loaded resource.

**The IPP on a grid.** The image processing pipeline is described as a solution of resources each one being represented by a sub-solution. Each resource contains the programs that it runs. Reaction rules between sub-solutions represent the coordination between resources. Figure 7 gives a possible initial state of the system running the pipeline. The initial solution contains several resources $\mathrm{R}_i$. A program is a pair of the form $name{:}\langle\ldots\rangle$ when the solution contains tasks to execute and data to process. The reaction $\mathrm{runFilter}$ executes the tasks stored in the solution $\mathrm{Tasks}$ of the program. The resource $\mathrm{R}_1$

$$\begin{array}{rl}
\textbf{let}\,\mathrm{runFilter} = & \textbf{replace}\ r{:}\langle \mathrm{Pipe}{:}\langle \mathrm{Tasks}{:}\langle f{:}x, \omega_R\rangle, \omega_1\rangle, \omega_2\rangle \\
 & \textbf{by}\ r{:}\langle \mathrm{Pipe}{:}\langle \mathrm{Tasks}{:}\langle \omega_R\rangle, (f\ x), \omega_1\rangle, \omega_2\rangle \\
\mathrm{applyF}_1 = & \textbf{replace}\ x{::}\mathrm{A},\ \mathrm{Tasks}{:}\langle \omega_R\rangle \\
 & \textbf{by}\ \mathrm{Tasks}{:}\langle \mathrm{F}_1{:}x, \omega_R\rangle \\
\mathrm{applyF}_2 = & \textbf{replace}\ x{::}\mathrm{B},\ \mathrm{Tasks}{:}\langle \omega_R\rangle \\
 & \textbf{by}\ \mathrm{Tasks}{:}\langle \mathrm{F}_2{:}x, \omega_R\rangle \\
 & \textbf{in}
\end{array}$$

$$\begin{array}{l}
\langle \mathrm{runFilter}, \\
\quad \mathrm{R}_1{:}\langle \mathrm{Pipe}{:}\langle \mathrm{applyF}_1, \mathrm{applyF}_2, \mathrm{Tasks}{:}\langle\rangle, \mathrm{i}_1, \mathrm{i}_2, \ldots\rangle\rangle,\ \mathrm{R}_2{:}\langle\rangle,\ \ldots\rangle
\end{array}$$

Figure 7
A possible initial state of a grid runtime system for the pipeline.

contains the initial state of the $\mathrm{Pipe}$ program. First, the rules $\mathrm{applyF}_1$ and $\mathrm{applyF}_2$ find possible tasks (i. e., filter applications) and store them as pairs $filter{:}image$ in the $\mathrm{Tasks}$ sub-solution. When the program becomes inert, the rule $\mathrm{runFilter}$ selects one of the stored tasks and runs it. The execution is finished when the top-level solution representing the system is inert. That is to say, when the solution $\mathrm{R}_1{:}\langle\ldots\rangle$ is inert and its sub-solution $\mathrm{Tasks}$ is empty (no task remains).

This example represents resources, programs, tasks and data as molecules. However, it does not take profit from the grid (the application remains on a single resource). We now coordinate the execution of the IPP to ensure two QoS:

- *efficiency*, by parallelizing the execution according to the availability of resources,
- *security*, by encrypting/decrypting messages sent over untrusted communication links.

**Parallelization.** To ensure efficiency, the pipeline is spread on resources and executed in parallel. Different migration rules can be added to the system. Figure 8 gives such a rule ($\mathrm{splitSPMD}$) and the rule $\mathrm{transfer}$ that moves messages (tasks and programs) between resources. In this part, messages

```
splitSPMD =
  let mergeSPMD =
        replace-one $r_1$:$\langle$Pipe:$\langle$applyF$_1 = f$, applyF$_2 = g$, Tasks:$\langle\rangle, \omega_1\rangle, \omega\rangle$,
                    $r_2$:$\langle$Pipe:$\langle$applyF$_1 = f'$, applyF$_2 = g'$, Tasks:$\langle\rangle, \omega_2\rangle, \omega'\rangle$
                 by $r_1$:$\langle$Pipe:$\langle$applyF$_1 = f$, applyF$_2 = g$, Tasks:$\langle\rangle, \omega_1\rangle, \omega\rangle$,
                    $r_2$:$\langle$ToSend:$r_1$:$\langle\omega_2\rangle, \omega'\rangle$
  in
  replace $r_1$:$\langle$Pipe:$\langle$applyF$_1 = f$, applyF$_2 = g$, Tasks:$\langle x\rangle, \omega_1\rangle, \omega'_1\rangle$,
          $r_2$:$\langle\omega_2\rangle$
     by $r_1$:$\langle$Pipe:$\langle$applyF$_1 = f$, applyF$_2 = g$, Tasks:$\langle x_1\rangle, \omega_1\rangle$,
              ToSend:$r_2$:(Pipe:$\langle$applyF$_1 = f$, applyF$_2 = g, x_2\rangle), \omega'_1\rangle$
          $r_2$:$\langle\omega_2\rangle$, mergeSPMD
      if $Unbalanced(r_1, r_2) \wedge (x_1, x_2) = BalancedSplit(r_1, r_2, x)$


  transfer = replace $r_1$:$\langle$ToSend:$r_2$:$m, \omega_1\rangle$, $r_2$:$\langle\omega_2\rangle$
                  by $r_1$:$\langle\omega_1\rangle$, $r_2$:$\langle m, \omega_2\rangle$
```

Figure 8
A possible distribution strategy of the pipeline.

could be left implicit but since they are central to the security QoS, we make them explicit using tuples of the form ToSend:$r$:$message$. The splitSPMD rule describes a possible distribution strategy of the pipeline. It splits the program in a Simple Program Multiple Data (SPMD) way. It is triggered by the predicate $Unbalanced(r_1, r_2)$ which is true when the load on $r_1$ is significantly higher than the load on $r_2$. In this case, a part of (or maybe the whole) program that is running on $r_1$ is migrated on $r_2$. This process is performed by the function $BalancedSplit(r_1, r_2, x)$ which takes a set of tasks ($filter$:$image$) and splits it into a set to be run on $r_1$ and a set to be run on $r_2$. The program Pipe and the images to be transfered are transformed into messages of the form ToSend:$r$:(Pipe:$\langle\ldots\rangle$). The reaction rule transfer will move the message to the destination resource ($r$). When two resources have completed their tasks (the solution Tasks is empty and no message is

$$\begin{aligned}\mathrm{secureTransfer} = \mathbf{replace}\ & r_1{:}\langle \mathrm{ToSend}{:}r_2{:}m, \omega_1\rangle,\ r_2{:}\langle\omega_2\rangle \\ \mathbf{by}\ & r_1{:}\langle\omega_1\rangle,\ r_2{:}\langle m,\omega_2\rangle \\ \mathbf{if}\ & \mathit{TrustedLink}(r_1, r_2) \\ & \quad \vee(\neg\,\mathit{TrustedLink}(r_1,r_2) \wedge \mathit{Encrypted}(m))\end{aligned}$$

$$\begin{aligned}\mathrm{encrypt} = \mathbf{replace}\ & r_1{:}\langle \mathrm{ToSend}{:}r_2{:}m, \omega_1\rangle \\ \mathbf{by}\ & r_1{:}\langle \mathrm{ToSend}{:}r_2{:}\mathit{Encrypt}(m), \omega_1\rangle \\ \mathbf{if}\ & \neg TrustedLink(r_1, r_2)\end{aligned}$$

$$\begin{aligned}\mathrm{decrypt} = \mathbf{replace}\ & r{:}\langle m, \omega\rangle \\ \mathbf{by}\ & r{:}\langle \mathit{Decrypt}(m), \omega\rangle \\ \mathbf{if}\ & \mathit{Encrypted}(m)\end{aligned}$$

Figure 9
Rules for the security QoS.

pending), the reaction rule mergeSPMD merges their result (a resource sends its result to the other). This rule will gather all partial results onto a single resource.

Other migration and coordination rules may be defined and added to the system. For example, we could add a rule splitMPMD that splits the program in a Multiple Program Multiple Data (MPMD) way.

All these rule sets specify coordination rules and can be added to the single resource system described in Figure 7. Splits and merges may be applied in any order on any resources. The system self-organizes to keep the load balanced between resources. At the end, the top-level solution (the whole system) is inert and the result is available on a single resource.

**Secure communications.** Here, the required QoS is to ensure secure communications between resources. Either the communication takes place on a trusted link or otherwise messages are encrypted. We replace the rule transfer by the three rules secureTransfer, encrypt and decrypt (see Figure 9). The rules encrypt and decrypt encrypts and decrypts messages when needed. The rule secureTransfer moves the message only if the communication link is trusted or if it has been encrypted (by the rule encrypt). Like in the mail system, these reaction rules for ensuring security are modular. They can be added to the system independently from the parallelization rules.

## 5 CONCLUSION

This article shows how the Chemical Programming paradigm can be used to describe self-organizing systems in a natural and elegant manner. In a first step, we introduced an expressive chemical language (called HOCL) whose higher-order properties allow the manipulation of programs. In HOCL, it becomes possible to write chemical reactions to create, delete or move around other chemical reactions. This feature is key in the description of self-organizing systems as shown in the two examples we have developed.

Self-organizing systems behave autonomously in order to maintain a predetermined quality of service which may be violated in certain circumstances. Very often, such violations may be dealt with by applying local corrections according to pre-defined rules. These rules are easily expressed as HOCL reactions. The mail system as well as the coordination rules for grids are non trivial examples which illustrate self-organizing chemical computations.

To the authors' knowledge, there has been no other attempt to program self-organizing systems within the chemical paradigm framework. The closest contributions use rule-based languages such as Prolog to specify expert systems whose knowledge is represented as a collection of facts and (rewrite) rules. Some attempts have been reported concerning development of self-organizing applications on grids. The Project AutoMate [13] aims at defining self-management applications through the concept of autonomic elements which exhibit a self-organizing behavior. P-Grid [1] is a self-organizing peer-to-peer system. Peers run agents that store and retrieve data in the grid in a complete decentralized and autonomic way. In organic grids [7], the organization of computation is based on the autonomous scheduling of strongly mobile agents on a peer-to-peer network.

The contribution presented here is part of a more general research program on the use of chemical languages as coordination languages for the description of grid systems and applications. The basic challenge consists in showing that HOCL allows a clean and elegant expression of features such as program mobility, load balancing, crash recovery, etc. Basically, the overall system is expressed as a "soup" (represented by a multiset) of resources such as processors, storage, communication links, etc. whose organization is described by appropriate reaction rules. Further work consists in designing generic chemical operations to express more easily coordination (resource manipulation, task and data migration, etc.) in grids. Hopefully, programming a grid will be as simple as specifying chemical solutions and reaction rules to describe how to distribute and execute them on a variety of resources.

## REFERENCES

[1] Kar Aberer. (2001). P-grid: A self-organizing access structure for p2p information systems. In *Proceedings of CoopIS 2001 conference*, number 2172 in LNCS. Springer-Verlag.

[2] Jean-Pierre Banâtre, Pascal Fradet, and Daniel Le Métayer. (2001). Gamma and the chemical reaction model: Fifteen years after. In *Multiset Processing*, volume 2235 of *LNCS*, pages 17–44. Springer-Verlag.

[3] Jean-Pierre Banâtre, Pascal Fradet, and Yann Radenac. (June 2004). Principles of chemical programming. In *Fifth International Workshop on Rule-Based Programming (RULE'04)*. Electronic Notes in Theoretical Computer Science 2005.

[4] Jean-Pierre Banâtre, Pascal Fradet, and Yann Radenac. (November 2005). Generalized multisets for chemical programming. Research Report 5743, INRIA.

[5] Jean-Pierre Banâtre and Daniel Le Métayer. (September 1986). A new computational model and its discipline of programming. Research Report 0566, INRIA.

[6] Jean-Pierre Banâtre and Daniel Le Métayer. (January 1993). Programming by multiset transformation. *Communications of the ACM (CACM)*, 36(1):98–111.

[7] Arjav J. Chakravarti, Gerald Baumgartner, and Mario Lauria. (2004). The organic grid: self-organizing computation on a peer-to-peer network. In *Proceedings of the Autonomic Computing Conference*, pages 96–103.

[8] Peter Dittrich, Jens Ziegler, and Wolfgang Banzhaf. (2001). Artificial chemistries – a review. *Artificial Life*, 7(3):225–275.

[9] Ian Foster and Carl Kesselman, editors. (1999). *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc.

[10] Chris Hankin, Daniel Le Métayer, and David Sands. (August 1992). A calculus of Gamma programs. In *Languages and Compilers for Parallel Computing, 5th International Workshop*, volume 757 of *LNCS*, pages 342–355. Springer-Verlag.

[11] Jeffrey Kephart and David Chess. (January 2003). The vision of autonomic computing. *IEEE Computer*.

[12] Manish Parashar and Salim Hariri. (2005). Autonomic computing: An overview. In Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors, *Unconventional Programming Paradigms*, volume 3566 of *LNCS*, pages 257–269. Springer.

[13] Manish Parashar, Hua Liu, Zhen Li, Vincent Matossian, Cristina Schmidt, Guangsen Zhang, and Salim Hariri. (2006). AutoMate: Enabling autonomic grid applications. In *Cluster Computing: The Journal of Networks, Software Tools, and Applications, Special Issue on Autonomic Computing*, volume 9. Kluwer Academic.