Classical Coordination Mechanisms in the Chemical Model

J.-P. Banâtre[*] P. Fradet[†] Y. Radenac[*]

In memory of Gilles Kahn

The essence of this paper stems from discussions that the first author (Jean-Pierre Banâtre) had with Gilles on topics related with programming in general and chemical programming in particular. Gilles liked the ideas behind the Gamma model [6] and the closely related Berry and Boudol's CHAM [7] as the basic principles are so simple and elegant. The last opportunity Jean-Pierre had to speak about these ideas to Gilles, was when he presented the LNCS volume devoted to the Unconventional Programming Paradigms workshop [1]. The 10 minutes appointment (at that time, he was CEO of INRIA) lasted a long time. Gilles was fine and in good humor, as often, and he was clearly happy to talk about a subject he loved. He spoke a lot about λ-calculus, the reduction principle, the β-reduction… a really great souvenir!

Abstract

Originally, the chemical model of computation has been proposed as a simple and elegant parallel programming paradigm. Data is seen as "molecules" and computation as "chemical reactions": if some molecules satisfy a predefined reaction condition, they are replaced by the "product" of the reaction. When no reaction is possible, a normal form is reached and the program terminates. In this paper, we describe classical coordination mechanisms and parallel programming models in the chemical setting. All these examples put forward the simplicity and expressivity of the chemical paradigm. We pay a particular attention to the chemical description of the simple and successful parallel computation model known as Kahn Process Networks.

[*]INRIA/IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France, {jbanatre,yradenac}@irisa.fr

[†]INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 Montbonnot, France Pascal.Fradet@inria.fr

1 Introduction

The Gamma formalism was proposed twenty years ago to capture the intuition of computation as the global evolution of a collection of atomic values interacting freely [6]. 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 or *inert* 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 multiset of comparable elements can be described by the reaction rule

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

meaning that any couple of elements x and y of the multiset such that x is greater or equal to y is replaced by x. 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, reactions can be performed in parallel.

Gamma can be formalized as a multiset AC-rewriting language. The Gamma formalism, and its derivative works as summarized in [2], is based on finite multisets of basic values. However, this basic concept can be extended by allowing elements of multisets to be reactions themselves (*higher-order multisets*), to have an infinite multiplicity (*infinite multisets*) and even to have a negative multiplicity (*hybrid multisets*). In [4], we have investigated these unconventional multiset structures (higher-order, infinite and hybrid multisets) and shown how they can be interpreted in a chemical programming framework. In particular, we have introduced the γ-calculus, a minimal higher-order calculus that summarizes the fundamental concepts of chemical programming. From this basic language, we have derived HOCL (the Higher Order Chemical Language), a programming language built by extending the γ-calculus with constants, operators, types and expressive patterns. The reflexive CHAM [10] is another approach to higher-order multiset programming.

The objective of this paper is to show how many well-known parallel mechanisms from basic mutual exclusion to Kahn Process Networks can be expressed in the same unified framework: the chemical model. This work illustrates one more time the expressivity of chemical languages. It also paves the way to formal comparisons of classical coordination structures and models.

2 The higher-order chemical language

The HOCL language [4] is a higher-order extension of Gamma based on the γ-calculus [3]. Here, we present briefly and informally the features of HOCL used in this article. The interested reader will find a more complete and formal presentation in [4].

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

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

that is, a multiset of constants, reaction rules and (sub-)solutions. The associativity and commutativity of the operator "," formalize the Brownian motion within a chemical solution. These laws can always be used to reorganize molecules in solutions. Atoms are either basic constants (integers, booleans, *etc.*), pairs (A_1:A_2), sub-solutions ($\langle M \rangle$) or reaction rules. A reaction rule is written

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

where P is a pattern which selects some atoms, C is the reaction condition and M the result of the reaction. If P matches atoms which satisfy C, they are replaced by M. For example,

$$\langle (\textbf{one}\, x{::}\text{Int}\, \textbf{by}\, x+1\, \textbf{if}\, x \text{ div } 2), 4, 9, 15 \rangle \longrightarrow_\gamma \langle 5, 9, 15 \rangle.$$

The pattern x::Int matches any integer, the condition imposes the integer to be even and the action replaces it by the next odd integer. In the rest of this article, we omit types in patterns when there is no ambiguity.

Such reaction rules are said to be *one-shot* since they are consumed when they react. In Gamma, rewrite rules were outside the multiset and remained as long as they could be applied. In HOCL, such recursive rules are called *n-shot* and, like in Gamma, there are written as

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

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 HOCL program computes the prime numbers lower than 10 using a version of the Eratosthenes' sieve:

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

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

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 content of a sub-solution only if it is inert. This important restriction allows to control the evaluation order in an otherwise highly non deterministic and 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 content matches the pattern P.

Rules can be named (or tagged) using the syntax

$$\text{name} = \textbf{replace}\, P\, \textbf{by}\, M\, \textbf{if}\, C.$$

Names are used to match and extract specific rules using the same syntax (name $= x$). We often use the **let** operator to name rules and assume that

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

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

We also often make use of the pattern ω which can match any molecule or nothing. This pattern is very convenient to extract elements from a solution.

Using all these features, the Eratosthenes' sieve can be rewritten in order to remove the n-shot reaction rule sieve at the end of the computation:

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

The reduction proceeds as follows:

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

The reaction rule clean cannot be applied until the sub-solution is inert. The rule sieve reacts until all primes are computed. Then, the one-shot rule clean extracts the prime numbers and suppresses the reaction rule sieve.

3 The basic coordination structures of HOCL

HOCL is a programming language that, as explained below, provides some primitive coordination structures: namely, parallel execution, mutual exclusion, the atomic capture and the serialization and parallelization of computations.

3.1 Parallel execution

When two reactions involve distinct tuples of elements, both reactions can occur at the same time.

For example, when computing the sum of a multiset of integers:

$$\langle 42, 6, 14, 5, 2, 8, 5, 42, 89, \text{add} = \textbf{replace}\, x, y\, \textbf{by}\, x + y \rangle$$

several reactions involving the rule add may occur at the same time provided that the couples of integers involved are distinct. Parallel execution relies on a fundamental property of HOCL: mutual exclusion.

3.2 Mutual exclusion

The mutual exclusion property states that a molecule cannot take part to several reactions at the same time. For example, several reactions can occur at the same time in the previous solution (*e.g.*, (42,89) at the same time as (5,5), *etc.*). Without mutual exclusion, the same integer could occur in several reactions at the same time. In this case, our previous program would not represent the sum of a multiset since, for example, 89 would be allowed to react with 2 and 6 and be replaced by 91 and 95.

3.3 Atomic capture

Another fundamental property of HOCL is the atomic capture. A reaction rule takes all its arguments atomically. Either all the required arguments are present or no reaction occurs. If all the required arguments are present, none of them may take part in another reaction at the same time.

Atomic capture is useful to express non blocking programs. For example, the famous dining philosophers problem can be expressed in HOCL as follows. Initially the multiset contains N forks (*i.e.,* N pairs Fork:1, ..., Fork:N) and the two following n-shot reaction rules eat and think:

$$\begin{aligned}
\text{eat} = \textbf{replace}\ & \text{Fork:}f_1,\ \text{Fork:}f_2 \\
\textbf{by}\ & \text{Phi:}f_1 \\
\textbf{if}\ & f_2 = f_1 + 1 \bmod N \\
\text{think} = \textbf{replace}\ & \text{Phi:}f \\
\textbf{by}\ & \text{Fork:}f,\ \text{Fork:}(f + 1 \bmod N) \\
\textbf{if}\ & \textbf{true}
\end{aligned}$$

The eat rule looks for two adjacent forks Fork:f_1 and Fork:f_2 with $f_2 = f_1 + 1 \bmod N$ and "produces" the eating philosopher Phi:f_1. This reaction relies on the atomic capture property: the two forks are taken simultaneously (atomicity) and this prevents deadlocks. The think rule "transforms" an eating philosopher into two available forks. This rule models the fact that any eating philosopher can be stopped non deterministically at anytime.

3.4 Serialization

A key motivation of chemical models in general, and HOCL in particular, is to be able to express programs without any artificial sequentiality (*i.e.,* sequentiality that is not imposed by the logic of the algorithm). However, even within this highly unconstrained and parallel setting, sequencing of actions can be expressed. Sequencing relies on the fact that a rule needing to access a sub-solution has to wait for its inertia. The reaction rule will react after (in sequence) all the reactions inside the sub-solution have completed.

The HOCL program that computes all the primes lower than a given integer N can be expressed by a sequence of actions that first computes the integers from 1 to N and then applies the rule sieve:

$$\langle\langle \text{iota}, \overline{N}\rangle, \text{thensieve}\rangle$$

where

$$\begin{aligned}\text{thensieve} &= \textbf{one}\langle \text{iota} = r, \overline{x}, \omega\rangle\ \textbf{by}\ \text{sieve}, \omega\\ \text{iota} &= \textbf{replace}\ \overline{x}\ \textbf{by}\ x, \overline{x-1}\ \textbf{if}\ x > 1\\ \text{sieve} &= \textbf{replace}\ x, y\ \textbf{by}\ x\ \textbf{if}\ x\ \text{div}\ y\end{aligned}$$

The rule iota generates the integers from N to 1 using the notation $\overline{x}$ to denote a distinguished (*e.g.*, tagged) integer. The one-shot rule thensieve waits for the inertia of the sub-solution. When it is inert, the generated integers are extracted and put next to the rule sieve (iota and the tagged integer $\overline{1}$ are removed). The wait for the inertia has serialized the iota and sieve operations.

Most of the existing chemical languages share these basic features. They all have conditional reactions with atomic capture of elements. On the other hand, they usually do not address fairness issues.

4 Classical coordination in HOCL

In this section, we consider well known coordination mechanisms and express them in HOCL. Most of them are communication patterns between sequential processes. We first show how to model such sequential processes as chemical solutions. Then, we propose a chemical version of communications using rendezvous, shared variables, Linda primitives and Petri nets.

4.1 Sequential processes in HOCL

In order to represent sequential and deterministic processes in the chemical model, we encode them at a fairly low level. For instance, the program counter and addresses of instructions will be represented explicitly. It should be clear that syntactic sugar could be used to express sequential processes (and many subsequent communication schemes) more concisely. However, for simplicity and uniformity reasons, we will stick to pure and basic HOCL programs.

A process is a solution made of:

- a local store storing variables represented by pairs of the form *name*:*value*;
- a code represented by a sequence of instructions encoded by pairs of the form *address*:*instruction* where *address* is an integer and *instruction* is a reaction rule to modify local variables;
- a program counter PC:*address* recording the next instruction to be executed.

A process is an inert solution that contains both the program to execute and its local store. A reaction rule, named run, executes the current instruction (*i.e.*, the instruction pointed to by PC) of processes.

For example, the process $P = \{\mathrm{TEMP} := \mathrm{X};\ \mathrm{X} := \mathrm{Y};\ \mathrm{Y} := \mathrm{TEMP};\}$ that swaps the content of two variables X and Y is represented by the sequence of instructions:

$$code \equiv \begin{cases} 1{:}\langle \mathrm{assign}(\mathrm{TEMP}, \mathrm{X})\rangle, \\ 2{:}\langle \mathrm{assign}(\mathrm{X}, \mathrm{Y})\rangle, \\ 3{:}\langle \mathrm{assign}(\mathrm{Y}, \mathrm{TEMP})\rangle \end{cases}$$

where $\mathrm{assign}(A, B)$ is the rule

$$\begin{aligned} \mathrm{assign}(A, B) \equiv\ & \textbf{one}\ A{:}a,\ B{:}b \\ & \textbf{by}\ A{:}b,\ B{:}b \end{aligned}$$

which performs the assignment $A := B$.

Assuming an initial store where the local variables X, Y and TEMP have the values 10, 23 and 0 respectively, the process P in its initial state is represented by the solution

$$\langle \mathrm{P}{:}\langle \mathrm{PC}{:}1, \mathrm{X}{:}10, \mathrm{Y}{:}23, \mathrm{TEMP}{:}0, code\rangle, \mathrm{run}\rangle$$

with

$$\begin{aligned} \mathrm{run} = \ & \textbf{replace}\ p{:}\langle \mathrm{PC}{:}a,\ a{:}\langle c\rangle,\ \omega\rangle \\ & \textbf{by}\ p{:}\langle \mathrm{PC}{:}(a+1),\ a{:}\langle c\rangle,\ c,\ \omega\rangle \end{aligned}$$

The run rule extracts the instruction pointed to by PC from the list of instructions and increments the program counter. The rule representing the instruction (*i.e.,* c) reacts and modifies the local store. When the solution representing the process is inert again, the run rule can be applied again. In our example, it is easy to convince oneself that the solution will be rewritten into the final inert solution

$$\langle \mathrm{P}{:}\langle \mathrm{PC}{:}4,\ \mathrm{X}{:}23,\ \mathrm{Y}{:}10,\ \mathrm{TEMP}{:}10,\ code\rangle,\ \mathrm{run}\rangle$$

Since there is no instruction at address 4, the run rule cannot react anymore. Other instructions are easily defined, for example:

- the neg rule

  $$\begin{aligned} \mathrm{neg}(A) \equiv\ & \textbf{one}\ A{:}a \\ & \textbf{by}\ A{:}(\mathrm{not}\ a) \end{aligned}$$

  corresponds to the statement $A := \mathrm{not}\ A$.

- the add rule

  $$\begin{aligned} \mathrm{add}(A, B, C) \equiv\ & \textbf{one}\ A{:}a,\ B{:}b,\ C{:}c \\ & \textbf{by}\ A{:}(b+c),\ B{:}b,\ C{:}c \end{aligned}$$

  corresponds to the statement $A := B + C$.

- the jmp instruction

  $$\begin{aligned} \mathrm{jmp}(b) \equiv\ & \textbf{one}\ \mathrm{PC}{:}a \\ & \textbf{by}\ \mathrm{PC}{:}b \end{aligned}$$

  sets PC to a new address.

- the ncondJmp instruction

$$\begin{aligned}\text{ncondJmp}(B, b) \equiv\ & \textbf{one}\ \text{PC}{:}a, B{:}k \\ & \textbf{by}\ B{:}k,\ PC{:}(\textbf{if}\ \ k\ \ \textbf{then}\ \ a\ \ \textbf{else}\ \ b)\end{aligned}$$

  sets PC to a new address or leaves it unchanged depending on the value of the boolean variable B.

The run rule can be placed into a solution with several processes like:

$$\langle \text{run},\ \text{P}_1{:}\langle \ldots \rangle, \ldots,\ \text{P}_n{:}\langle \ldots \rangle \rangle$$

In absence of coordination rule, run will rewrite all processes step by step potentially in parallel or non-deterministically. We now describe coordination (communication, synchronization) instructions between two (or more) processes. Contrary to standard imperative instructions which are of the form $a{:}\langle i \rangle$ (*i.e.,* within a solution), coordination instructions will be of the form $a{:}i{:}x{:}y$ (*i.e.,* in tuples). They cannot be executed by the run rule. The coordination mechanism relies on specific reaction rules taking as parameters all actors (processes, shared variables, queues, *etc.*) involved in the interaction.

4.2 Rendezvous in HOCL

We consider concurrently executing processes communicating by atomic, instantaneous actions called "rendezvous" (or sometimes, "synchronous message passing"). If two processes are to communicate, and one reaches the point at which it is ready to communicate first, then it stalls until the other process is ready as well. The exchange (communication) is atomic in that it is initiated and completed in a single uninterruptable step. Examples of rendezvous models include Hoare's communicating sequential processes (CSP) [11] and Milner's calculus of communicating systems (CCS) [13].

Rendezvous models are particularly well-suited to applications where resource sharing is a key element, such as client-server database models and multitasking or multiplexing of hardware resources. A key feature of rendezvous-based models is their ability to cleanly model nondeterminate interactions.

In HOCL, the rendezvous is represented as an atomic capture of two processes in a particular state. The sender should be ready to send something and the receiver should be ready to receive. The rule runRdV, below, implements a rendezvous communication in one atomic step:

$$\begin{aligned}\text{runRdV} = \textbf{replace}\ & p_1{:}\langle \text{PC}{:}a,\ a{:}\text{send}{:}p_2{:}x,\ x{:}k,\ \omega_1 \rangle, \\ & p_2{:}\langle \text{PC}{:}b,\ b{:}\text{recv}{:}p_1{:}y,\ y{:}l,\ \omega_2 \rangle \\ \textbf{by}\ & p_1{:}\langle \text{PC}{:}(a+1),\ a{:}\text{send}{:}p_2{:}x,\ x{:}k,\ \omega_1 \rangle, \\ & p_2{:}\langle \text{PC}{:}(b+1),\ b{:}\text{recv}{:}p_1{:}y,\ y{:}k,\ \omega_2 \rangle\end{aligned}$$

The communication between two processes p_1 and p_2 takes place when the active instruction of p_1 is of the form send:$p_2{:}x$ ("send the value of variable x to p_2") and the active instruction of p_2 is of the form recv:$p_1{:}y$ ("place the value received from p_1 in variable y"). The value of x is placed into y and both program counters are incremented.

Typically, a collection of sequential processes communicating by rendezvous will be represented by a solution of the form:

$$\langle \text{run},\ \text{runRdV},\ \text{P}_1{:}\langle\ldots\rangle, \ldots,\ \text{P}_n{:}\langle\ldots\rangle\rangle$$

The reaction run will execute (in parallel and non-deterministically) processes unwilling to communicate. At the same time, runRdV will execute (potentially several) pairs of processes waiting for a rendezvous.

4.3 Shared variables

Local variables are inside the solution representing the associated process. Shared variables are represented by data in the top-level solution containing the processes. For example, the shared variable X whose value is 3 is represented as:

$$\langle \ldots,\ \text{P}_i{:}\langle\ldots\rangle,\ \ldots, \text{X}{:}3\rangle$$

A shared variable is manipulated using two instructions:

- a:Wshare:X:Y which writes the value of the local variable Y in the shared variable X;
- a:Rshare:X:Y which reads the value of the shared variable Y and stores it in the local variable X.

The associated reaction rules are

$$\begin{array}{rl}
\text{runWshare} = \textbf{replace} & p{:}\langle \text{PC}{:}a,\ a{:}\text{Wshare}{:}x{:}y,\ y{:}k,\ \omega\rangle, \\
& x{:}l \\
\textbf{by} & p{:}\langle \text{PC}{:}(a+1),\ a{:}\text{Wshare}{:}x{:}y,\ y{:}k,\ \omega\rangle, \\
& x{:}k \\
\text{runRshare} = \textbf{replace} & p{:}\langle \text{PC}{:}a,\ a{:}\text{Rshare}{:}x{:}y,\ x{:}k,\ \omega\rangle, \\
& y{:}l \\
\textbf{by} & p{:}\langle \text{PC}{:}(a+1),\ a{:}\text{Wshare}{:}x{:}y,\ x{:}l,\ \omega\rangle, \\
& y{:}l
\end{array}$$

To let processes communicate using shared variables A and B, the system is represented by a solution of the form:

$$\langle \text{run},\ \text{runWshare},\ \text{runRshare},\ \text{P}_1{:}\langle\ldots\rangle, \ldots,\ \text{P}_n{:}\langle\ldots\rangle,\ \text{A}{:}0,\ \text{B}{:}841\rangle$$

Again, the atomic capture of reaction rules is key to ensure the atomicity of the operations on shared variables. If several processes want to write the same shared variable concurrently, an ordering will be non-deterministically chosen.

4.4 Linda primitives in HOCL

The Linda model of communication [8] has mainly two operations, *out* and *in*, that modify a unique data structure called a tuple space. The operation $out(X_1{:}\cdots{:}X_n)$ stores a tuple $l_1{:}\cdots{:}l_n$ (where l_i is the value of the variable X_i) in the tuple space. The operation $in(X_1{:}\cdots{:}X_n)$ removes a tuple that matches the given tuple pattern. The tuple pattern contains variable names or constants. A corresponding tuple must be the same size and the constants must concur. Then, the ith variable X_i is assigned to the ith value. In HOCL these are implemented by the two following reaction rules:

$$\begin{aligned} \text{runLindaOut} = \textbf{replace}\ & p{:}\langle \text{PC}{:}a,\ a{:}\text{out}{:}r,\ \omega_P\rangle \\ \textbf{by}\ & p{:}\langle \text{PC}{:}a,\ a{:}\overline{\text{out}}{:}r,\ \omega_P\rangle, \\ & r \end{aligned}$$

$$\begin{aligned} \text{runLindaIn} = \textbf{replace}\ & p{:}\langle \text{PC}{:}a,\ a{:}\text{in}{:}r,\ \omega_P\rangle \\ \textbf{by}\ & p{:}\langle \text{PC}{:}a,\ a{:}\overline{\text{in}}{:}r,\ \omega_P\rangle, \\ & r \end{aligned}$$

In both rules, the variable r stands for a one-shot rule that is extracted to be executed. In a out-rule, r has the following form:

$$\begin{aligned} \textbf{one}\ & p{:}\langle \text{PC}{:}a,\ a{:}\overline{\text{out}}{:}r,\ X_1{:}l_1,\ldots,\ X_n{:}l_n,\ \omega_p\rangle, \\ & \text{TS}{:}\langle\omega\rangle \\ \textbf{by}\ & p{:}\langle \text{PC}{:}(a+1),\ a{:}\text{out}{:}r,\ X_1{:}l_1,\ldots,\ X_n{:}l_n,\ \omega_p\rangle, \\ & \text{TS}{:}\langle l_1{:}\ldots{:}l_n,\ \omega\rangle \end{aligned}$$

Conversely, in a in-rule, r has the following form:

$$\begin{aligned} \textbf{one}\ & p{:}\langle \text{PC}{:}a,\ a{:}\overline{\text{in}}{:}r,\ X_1{:}l'_1,\ldots,\ X_n{:}l'_n,\ \omega_p\rangle, \\ & \text{TS}{:}\langle l_1{:}\ldots{:}l_n,\ \omega\rangle \\ \textbf{by}\ & p{:}\langle \text{PC}{:}(a+1),\ a{:}\text{in}{:}r,\ X_1{:}l_1,\ldots,\ X_n{:}l_n,\ \omega_p\rangle, \\ & \text{TS}{:}\langle\omega\rangle \end{aligned}$$

The in and out operations are implemented in a two-step fashion. In the first step, the corresponding rule is extracted. The instruction name in or out is converted to $\overline{\text{in}}$ or $\overline{\text{out}}$ to prevent a second shot, and the PC is not incremented. In the second step, the extracted one-shot rule looks for all the data it needs inside the process p (read or write variables) and inside the tuple space TS (matched tuple). It increments the PC and resets the instruction name to in or out.

The extracted one-shot rule corresponding to a in-rule is blocked until a tuple matching $l_1{:}\ldots{:}l_n$ is found in the considered tuple space, and so the corresponding process is blocked too. The pattern $l_1{:}\ldots{:}l_n$ may contain variables and constants (*e.g.*, $1{:}x$).

Consider the server process of the Server-Clients example from [8]. A server treats a stream of requests encoded as a triple ("request", *index*, *data*) in the tuple space. Starting from index 1, the server extracts such a triple from the tuple space, assigns *data* to its local variable *req*, treats the request and produces the result as a triple (“response”, *index*, *data*) in the tuple space. The index is then incremented to treat the next request. The C-Linda code implementing the server is shown in Figure 1.

```
server()
{
    int index = 1;
    ...
    while (1) {
         in("request", index, ? req);
         ...
         out("response", index++, response);
    }
}
```

Figure 1: Server in C-Linda

Figure 2 shows the encoding of this program into HOCL. The sequential code of the process is similar. The in and out operations are encoded using the two one-shot rules *request* and *response*. The rule *request* extracts a tuple of the form (“request”:i:X) where i is the current index and assigns the data X to the local variable req. The rule *response* stores a tuple of the form (“response”:i:X) where X is the value of the local variable req and increments the current index i. In both cases, the program counter is incremented and the in and out instructions are reset to an executable form.

4.5 Petri nets in HOCL

A Petri net [14] consists in places and transitions. A place may contain any number of tokens. A transition takes some tokens from its input places and produces tokens in its output places.

The notions of places and transitions are very close to the chemical notions of solutions and reaction rules. We represent a place as a solution of tokens $\langle \mathrm{Tok}, \mathrm{Tok}, \ldots \rangle$ and transitions as reaction rules rewriting atomically input and output places. Places and transitions are named P_i and t_j respectively. For example, a transition t_1 taking two tokens from P_1 and producing one token into P_2 is represented by the rule

$$\begin{array}{c} \mathrm{t}_1 = \textbf{replace } \mathrm{P}_1{:}\langle \mathrm{Tok}, \mathrm{Tok}, \omega_1 \rangle,\ \mathrm{P}_2{:}\langle \omega_2 \rangle \\ \textbf{by } \mathrm{P}_1{:}\langle \omega_1 \rangle,\ \mathrm{P}_2{:}\langle \mathrm{Tok}, \omega_2 \rangle \end{array}$$

The main issue is that two transitions consuming tokens in the same place (that has enough tokens) cannot be fired simultaneously since a sub-solution can only take part to one reaction at a time.

An alternative encoding, that allows two transitions with common input places to be fired concurrently is to represent a token in place P_i as the constant

$$
\begin{array}{l}
\langle \text{run},\ \text{runLindaIn},\ \text{runLindaOut},\ \text{TS}{:}\langle\rangle, \\
\ \ \text{Server}{:}\langle\ \begin{array}[t]{l} \text{index}{:}1, \\ \text{PC}{:}0, \\ \text{req}{:}0, \\ 0{:}\text{in}{:}\mathit{request}, \\ \ldots \\ n{:}\text{out}{:}\mathit{response}, \\ (n+1){:}\text{jmp}(0) \end{array} \\
\ \ \rangle,\ \ \ldots \\
\rangle
\end{array}
$$

$$
\begin{array}{lrl}
\text{with} & \mathit{request} = & \textbf{one}\ \text{Server}{:}\langle \text{PC}{:}a,\ a{:}\overline{\text{in}}{:}r,\ \text{index}{:}i,\ \text{req}{:}j,\ \omega_p\rangle, \\
 & & \quad\ \ \text{TS}{:}\langle(\text{``request''}{:}i{:}X),\ \omega\rangle \\
 & & \ \ \textbf{by}\ \text{Server}{:}\langle \text{PC}{:}(a+1),\ a{:}\text{in}{:}r,\ \text{index}{:}i,\ \text{req}{:}X,\ \omega_p\rangle, \\
 & & \quad\ \ \text{TS}{:}\langle\omega\rangle \\
 & & \\
 & \mathit{response} = & \textbf{one}\ \text{Server}{:}\langle \text{PC}{:}a,\ a{:}\overline{\text{out}}{:}r,\ \text{index}{:}i,\ \text{req}{:}j,\ \omega_p\rangle, \\
 & & \quad\ \ \text{TS}{:}\langle\omega\rangle \\
 & & \ \ \textbf{by}\ \text{Server}{:}\langle \text{PC}{:}(a+1),\ a{:}\text{out}{:}r,\ \text{index}{:}(i+1),\ \text{req}{:}X,\ \omega_p\rangle, \\
 & & \quad\ \ \text{TS}{:}\langle(\text{``response''}{:}i{:}X),\ \omega\rangle
\end{array}
$$

Figure 2: Encoding of the C-Linda server into HOCL

P_i. All tokens are placed in the top-level solution and the previous transition t_1 would be expressed as:

$$\mathrm{t}_1 = \textbf{replace}\,\mathrm{P}_1, \mathrm{P}_1\ \textbf{by}\,\mathrm{P}_2$$

The drawback of this encoding is to prevent the expression of inhibitor arcs. An inhibitor arc from place P to t enables the transition t to fire only if no tokens are in the place P. Testing the absence of elements with the second encoding is a global operation which cannot be expressed as a single atomic rule. With the first encoding, inhibitor arcs can be encoded easily by testing whether a sub-solution is empty. For example:

$$
\begin{array}{rl}
\mathrm{t}_2 = \textbf{replace} & \mathrm{P}_1{:}\langle \text{Tok}, \omega_1\rangle,\ \mathrm{P}_2{:}\langle\rangle,\ \mathrm{P}_3{:}\langle\omega_3\rangle \\
\textbf{by} & \mathrm{P}_1{:}\langle\omega_1\rangle,\ \mathrm{P}_2{:}\langle\rangle,\ \mathrm{P}_3{:}\langle \text{Tok}, \omega_3\rangle
\end{array}
$$

The Petri net shown in Figure 3 (taken from [14]) represents a readers-writers synchronization, where the k tokens in place P_1 represent k processes which may read and write in a shared memory represented by place P_3. Up to k process may be reading concurrently, but when one process is writing, no other process can be reading or writing.

The encoding of that Petri net in HOCL is the solution given in Figure 4. Tokens are represented by constants, and the four transitions are represented by four n-shot rules. For instance, the rule t2 takes and removes one token P1 and k tokens P3, and generates one token P4.

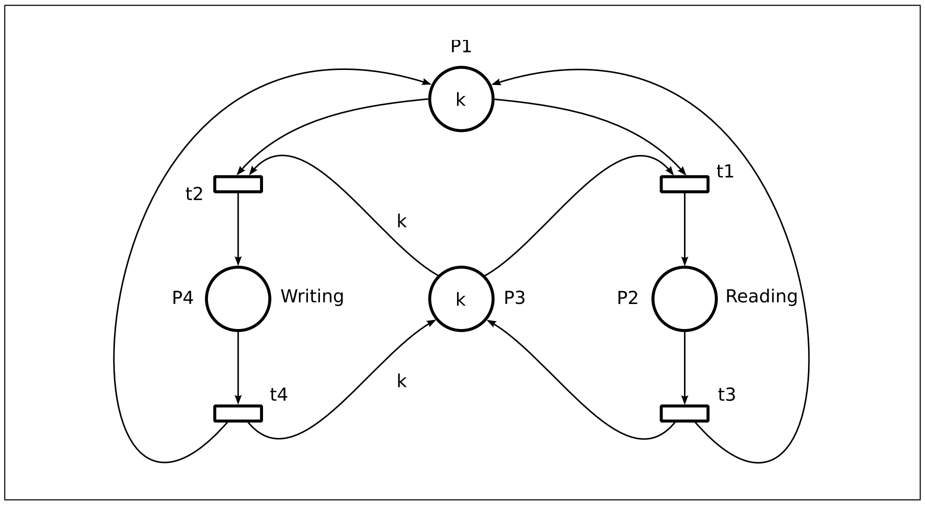

Figure 3: Petri net of a readers-writers synchronization

$$
\begin{array}{l}
\textbf{let}\ \text{t1} = \textbf{replace}\ \text{P1}, \text{P3}\ \textbf{by}\ \text{P2}\ \textbf{in} \\
\textbf{let}\ \text{t2} = \textbf{replace}\ \text{P1}, \underbrace{\text{P3}, \ldots,\ \text{P3}}_{k}\ \textbf{by}\ \text{P4}\ \textbf{in} \\
\textbf{let}\ \text{t3} = \textbf{replace}\ \text{P2}\ \textbf{by}\ \text{P1}, \text{P3}\ \textbf{in} \\
\textbf{let}\ \text{t4} = \textbf{replace}\ \text{P4}\ \textbf{by}\ \text{P1}, \underbrace{\text{P3}, \ldots,\ \text{P3}}_{k}\ \textbf{in} \\
\langle \text{t1},\ \text{t2},\ \text{t3},\ \text{t4},\ \text{P1},\ \underbrace{\text{P3}, \ldots,\ \text{P3}}_{k} \rangle
\end{array}
$$

Figure 4: Encoding of the Petri net in HOCL

5 Kahn Process Networks in a chemical setting

The Kahn Process Network (KPN) model of computation [12] assumes a network of concurrent autonomous and deterministic processes that communicate over unbounded FIFO channels. Communication is asynchronous and point to point; it is based on a blocking read primitive and a non-blocking send primitive. Each process in the network is specified as a sequential program that executes concurrently with other processes. A KPN is deterministic, meaning that the result of the computation is independent of its schedule.

5.1 A simple example

Gilles Kahn gave in [12] an example made out of the three following sequential processes:

```
Process f(integer in U,V; integer out W) ;
  Begin integer I ; logical B ;
    B := true ;
    Repeat Begin
      I := if B then wait(U) else wait(V) ;
      print (I) ;
      send I on W ;
      B := not B ;
    End ;
  End ;

Process g(integer in U ; integer out V, W ) ;
  Begin integer I ; logical B ;
    B := true ;
    Repeat Begin
      I := wait(U) ;
      if B then send I on V else send I on W ;
      B := not B ;
    End ;
  End ;

Process h(integer in U ; integer out V; integer INIT ) ;
  Begin integer I ;
    Repeat Begin
      I := wait(U) ;
      send I on V ;
    End ;
  End ;
```

A process writes the value of a local variable `X` on a channel `C` using the command `send X on C`. It reads the channel `C` and stores the value in the local variable `X` using the command `X := wait(C)`.

The network is built using one instance of each process `f` and `g` and two instances of process `h` (with its third parameter set to 0 and 1) and connecting them using the FIFO channels `X`, `Y`, `Z`, `T1` and `T2`. Using `par` for the parallel composition, the network is specified as

```
f(Y,Z,X) par g(X,T1,T2) par h(T1,Y,0) par h(T2,Z,1) ;
```

A graphical representation of the complete network is given by Figure 5 where nodes represent processes and arcs communication channels between processes.

5.2 Chemical queues

In KPNs, communication between processes uses queues to implement asynchronous communication. In HOCL, a queue can be represented by a solution of pairs *rank*:*value*. That solution includes also two counters CW (resp. CR) storing the rank to write (resp. to read) a value. Using the operation `send`, a producer adds a value at the rank CW and increments the counter CW. Using

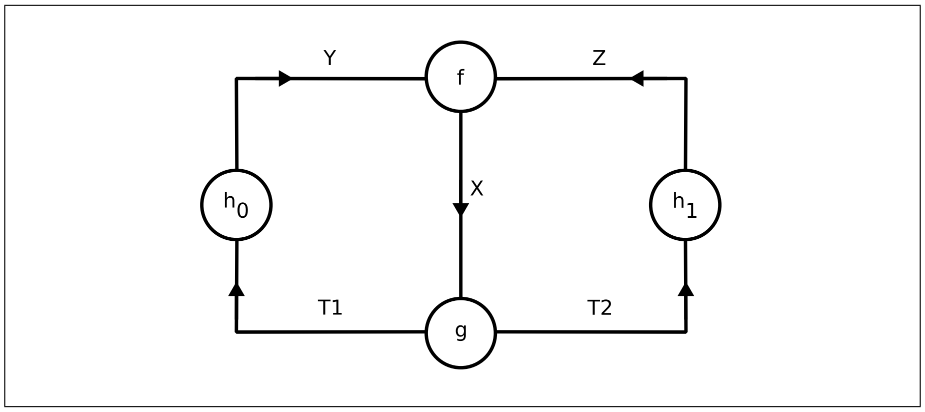

Figure 5: Graphical representation of the KPN example

the operation `wait`, a consumer takes the value at rank CR and increments the counter CR. The counters CW and CR are always present in the solution representing the queue. Initially, the queue is empty and both counters are equal.

The `send X on C` operation is represented in HOCL by the instruction a:send:C:X and `X := wait(C)` by a:wait:C:X. The corresponding rules are:

$$\begin{aligned}\text{runSend} = \textbf{replace}\ & p{:}\langle \text{PC}{:}a,\ a{:}\text{send}{:}q{:}u,\ u{:}k,\ \omega\rangle,\\ & q{:}\langle \text{CW}{:}cw,\ \omega_Q\rangle\\ \textbf{by}\ & p{:}\langle \text{PC}{:}(a+1),\ a{:}\text{send}{:}q{:}u,\ u{:}k,\ \omega\rangle,\\ & q{:}\langle \text{CW}{:}(cw+1),\ cw{:}k,\ \omega_Q\rangle\end{aligned}$$

$$\begin{aligned}\text{runWait} = \textbf{replace}\ & p{:}\langle \text{PC}{:}a,\ a{:}\text{wait}{:}q{:}u,\ u{:}k,\ \omega\rangle,\\ & q{:}\langle \text{CR}{:}cr,\ cr{:}l,\ \omega_Q\rangle\\ \textbf{by}\ & p{:}\langle \text{PC}{:}(a+1),\ a{:}\text{wait}{:}q{:}u,\ u{:}l,\ \omega\rangle,\\ & q{:}\langle \text{CR}{:}(cr+1),\ \omega_Q\rangle\end{aligned}$$

The instructions send and wait can be seen as a rendezvous between a process p and a queue q. That implementation supports unbounded queues but does not allow a producer and a consumer to access the same queue at the same time. We present other possible encoding at the end of the section.

5.3 Chemical KPNs

Typically, a KPN is represented by several sequential processes and queues in the main solution. The solution contains also the three reaction rules run, runSend and runWait which can be performed at the same time (as long as they do not involve the same queues or processes). Chemical reactions make the network evolve according a unspecified, non-deterministic and parallel schedule. The only synchronization constraints are enforced by queues. Of course, even if there is much potential interleaving, the functional semantics of chemical KPNs is deterministic.

The example of section 5.1 can be written in HOCL using the previous encoding for sequential commands (see section 4.1) and queues as follows:

$$\begin{array}{lll}
\mathit{processF}(u,v,w) \equiv & \mathit{processG}(u,v,w) \equiv & \mathit{processH}(u,v,\mathit{init}) \equiv \\
\quad \langle \mathrm{PC}{:}0, & \quad \langle \mathrm{PC}{:}0, & \quad \langle \mathrm{PC}{:}0, \\
\quad\ \ \mathrm{I}{:}0,\ \ \mathrm{B}{:}\mathrm{true}, & \quad\ \ \mathrm{I}{:}0, \mathrm{B}{:}\mathrm{true}, & \quad\ \ \mathrm{I}{:}0, \\
\quad\ \ 0{:}\langle \mathrm{ncondJmp}(\mathrm{B},3)\rangle, & \quad\ \ 0{:}\mathrm{wait}{:}u{:}\mathrm{I}, & \quad\ \ 0{:}\mathrm{send}{:}v{:}\mathit{init}, \\
\quad\ \ 1{:}\mathrm{wait}{:}u{:}\mathrm{I}, & \quad\ \ 1{:}\langle \mathrm{ncondJmp}(\mathrm{B},4)\rangle, & \quad\ \ 1{:}\mathrm{wait}{:}u{:}\mathrm{I}, \\
\quad\ \ 2{:}\langle \mathrm{jmp}(4)\rangle, & \quad\ \ 2{:}\mathrm{send}{:}v{:}\mathrm{I}, & \quad\ \ 2{:}\mathrm{send}{:}v{:}\mathrm{I}, \\
\quad\ \ 3{:}\mathrm{wait}{:}v{:}\mathrm{I}, & \quad\ \ 3{:}\langle \mathrm{jmp}(5)\rangle, & \quad\ \ 3{:}\langle \mathrm{jmp}(1)\rangle\rangle \\
\quad\ \ 4{:}\langle \mathrm{print}, \mathrm{I}\rangle, & \quad\ \ 4{:}\mathrm{send}{:}w{:}\mathrm{I}, & \\
\quad\ \ 5{:}\mathrm{send}{:}w{:}\mathrm{I}, & \quad\ \ 5{:}\langle \mathrm{neg}(\mathrm{B})\rangle, & \\
\quad\ \ 6{:}\langle \mathrm{neg}(\mathrm{B})\rangle, & \quad\ \ 6{:}\langle \mathrm{jmp}(0)\rangle\rangle & \\
\quad\ \ 7{:}\langle \mathrm{jmp}(0)\rangle\rangle & &
\end{array}$$

The processes have local variables (I and B, or just I) and their program counter set to 0. The network of Figure 5 is represented by the following solution:

$$\begin{array}{l}
\langle \mathit{processF}(\mathrm{Y},\mathrm{Z},\mathrm{X}),\ \mathit{processG}(\mathrm{X},\mathrm{T1},\mathrm{T2}), \\
\ \mathit{processH}(\mathrm{T1},\mathrm{Y},0),\ \mathit{processH}(\mathrm{T2},\mathrm{Z},1), \\
\ \mathrm{Y}{:}\langle \mathrm{CW}{:}0, \mathrm{CR}{:}0\rangle,\ \mathrm{Z}{:}\langle \mathrm{CW}{:}0, \mathrm{CR}{:}0\rangle,\ \mathrm{X}{:}\langle \mathrm{CW}{:}0, \mathrm{CR}{:}0\rangle, \\
\ \mathrm{T1}{:}\langle \mathrm{CW}{:}0, \mathrm{CR}{:}0\rangle,\ \mathrm{T2}{:}\langle \mathrm{CW}{:}0, \mathrm{CR}{:}0\rangle, \\
\ \mathrm{run},\ \mathrm{runWait},\ \mathrm{runSend}\rangle
\end{array}$$

The solution is made of four instances of processes, five queues (initially empty), and the reaction rules implementing sequential execution (run) and communication (runWait, runSend). Our implementation remains close to the original example, the key difference being that sequential execution and communications are made explicit by reaction rules.

5.4 Other implementations of queues

We conclude our study of KPNs by presenting some other possible implementations of queues in the same chemical framework.

Bounded queues

A bounded queue is a solution $q{:}n{:}\langle \ldots \rangle$ tagged by its name q and maximum size n. If a queue is bounded, the corresponding counters are incremented modulo its maximum size n. The rule runSendB is now a blocking primitive since it may only react when the queue is not full:

$$\begin{array}{rl}
\mathrm{runSendB} = \textbf{replace} & p{:}\langle \mathrm{PC}{:}a,\ a{:}\mathrm{send}{:}q{:}u,\ u{:}k,\ \omega\rangle, \\
& q{:}n{:}\langle \mathrm{CW}{:}cw,\ \mathrm{CR}{:}cr,\ \omega_Q\rangle \\
\textbf{by} & p{:}\langle \mathrm{PC}{:}(a+1),\ a{:}\mathrm{send}{:}q{:}u,\ u{:}k,\ \omega\rangle, \\
& q{:}n{:}\langle \mathrm{CW}{:}(cw+1) \bmod n,\ \mathrm{CR}{:}cr,\ cw{:}k,\ \omega_Q\rangle \\
\textbf{if} & (cw+1) \bmod n \neq cr
\end{array}$$

$$\begin{aligned}\text{runWaitB} = \textbf{replace}\ & p{:}\langle \text{PC}{:}a,\ a{:}\text{wait}{:}q{:}u,\ u{:}k,\ \omega\rangle,\\ & q{:}n{:}\langle \text{CR}{:}cr,\ cr{:}l,\ \omega_Q\rangle\\ \textbf{by}\ & p{:}\langle \text{PC}{:}(a+1),\ a{:}\text{wait}{:}q{:}u,\ u{:}l,\ \omega\rangle,\\ & q{:}n{:}\langle \text{CR}{:}(cr+1) \bmod n,\ \omega_Q\rangle\end{aligned}$$

The queue is full when the next free rank (CW + 1) is equal modulo n to CR. In this case, the send operation blocks.

Non exclusive queues

When a queue is represented by a sub-solution, it cannot react simultaneously with several processes, especially between one producer and one consumer. In this case, using independent atoms to represent a queue solves this problem and the representations of all queues are mixed together. The two counters of a queue are tagged by the name of the queue (CW:q:cw, CR:q:cr). The values are also tagged and are represented as triples q:r:v (name of queue, rank, value). The reaction rules implementing writing and reading become:

$$\begin{aligned}\text{runSendNEx} = \textbf{replace}\ & p{:}\langle \text{PC}{:}a,\ a{:}\text{send}{:}q{:}u,\ u{:}k,\ \omega\rangle,\\ & \text{CW}{:}q{:}cw\\ \textbf{by}\ & p{:}\langle \text{PC}{:}(a+1),\ a{:}\text{send}{:}q{:}u,\ u{:}k,\ \omega\rangle,\\ & \text{CW}{:}q{:}(cw+1),\ q{:}cw{:}k\end{aligned}$$

$$\begin{aligned}\text{runWaitNEx} = \textbf{replace}\ & p{:}\langle \text{PC}{:}a,\ a{:}\text{wait}{:}q{:}u,\ u{:}k,\ \omega\rangle,\\ & \text{CR}{:}q{:}cr,\ q{:}cr{:}l\\ \textbf{by}\ & p{:}\langle \text{PC}{:}(a+1),\ a{:}\text{wait}{:}q{:}u,\ u{:}l,\ \omega\rangle,\\ & \text{CR}{:}q{:}(cr+1)\end{aligned}$$

These operations are no longer rendezvous with the queue but instead with the counter CW for runSendNEx and the counter CR and an individual value for runWaitNEx. The same queue can be written and read at the same time as long as it has at least one value.

6 Conclusion

Originally, the Gamma formalism was invented as a basic paradigm for parallel programming [6]. It was proposed to capture the intuition of a computation as the global evolution of a collection of atomic values evolving freely. Gamma appears as a very high level language which allows programmers to describe programs in a very abstract way, with minimal constraints and no artificial sequentiality. In fact, from experience, it is often much harder to write a sequential program in Gamma than a parallel one. Later, it became clear that a necessary extension to this simple formalism was to allow elements of a multiset to be Gamma programs themselves, thus introducing higher-order. This lead to the HOCL language used in this paper.

The idea behind the present paper was to show how traditional coordination mechanisms can be readily described in a chemical setting. Basically, the chemical paradigm (as introduced in HOCL) offers four basic properties: mutual exclusion, atomic capture, parallelization and serialization. We have exploited these properties in order to give a chemical expression of well known coordination schemes. After presenting how to encode sequential processes as chemical solutions, we have expressed the CSP rendezvous, shared variables, Linda's primitives, Petri nets and Kahn Process Networks in HOCL. All these examples put forward the simplicity and expressivity of the chemical paradigm. A natural research direction would be to complete and use these descriptions in the same chemical framework to compare and classify the different coordination schemes.

The chemical paradigm has been used in several other areas. For example, an operating system kernel [15] has been specified in Gamma and proved correct in a framework inspired by the Unity logic [9]. The system is represented as a collection of quadruples (P_i, S_i, M_i, C_i) where S_i, M_i, and C_i represent respectively the state, the mailbox and the channel associated with process P_i. The P_i's are functions called by the system itself. The system includes rules such as

$$\textbf{replace}(P_i, S_i, M_i, C_i)\,\textbf{by}(P_i', S_i', M_i', C_i')\,\textbf{if}\,\mathit{Ready}(P_i, S_i)$$

An important aspect of this work is the derivation of a file system (written in Gamma) by successive refinements from a temporal logic specification.

More recently, we have used HOCL to specify autonomic systems [5]. Such 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. These corrections are easily expressed as independent HOCL reaction rules. Comparisons with models related to HOCL and the underlying γ-calculus may be found in [4]. We do not come back on these comparisons here.

As a final comment, let us point out that, unlike Gamma, HOCL allows reaction rules (programs) to be elements of multisets, to be composed by taking (resp. returning) reactions as parameters (resp. result) and to be recursive (as expressed by the **replace** rule). In that sense, it complies with the principle stated in the conclusion of Gilles's paper [12]:

> "A *good* concept is one that is closed under arbitrary composition and under recursion."

Acknowledgments

Thanks are due to the anonymous reviewers for their comments and suggestions.

References

[1] Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors. *Unconventional Programming Paradigms (UPP'04)*, vol-

ume 3566 of *LNCS*, Revised Selected and Invited Papers of the International Workshop, 2005. Springer-Verlag.

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

[3] Jean-Pierre Banâtre, Pascal Fradet, and Yann Radenac. Principles of chemical programming. In S. Abdennadher and C. Ringeissen, editors, *Proceedings of the 5th International Workshop on Rule-Based Programming (RULE 2004)*, volume 124 of *ENTCS*, pages 133–147. Elsevier, June 2005.

[4] Jean-Pierre Banâtre, Pascal Fradet, and Yann Radenac. Generalised multisets for chemical programming. *Mathematical Structures in Computer Science*, 16(4):557–580, August 2006.

[5] Jean-Pierre Banâtre, Pascal Fradet, and Yann Radenac. Programming self-organizing systems with the higher-order chemical language. *International Journal of Unconventional Computing*, 2007.

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

[7] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

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

[9] K. Mani Chandy and Jayadev Misra. *Parallel Program Design : A Foundation.* Addison-Wesley, 1988.

[10] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM Press, 1996.

[11] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[12] Gilles Kahn. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[13] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[14] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[15] Héctor Ruiz Barradas. *Une approche à la dérivation formelle de systèmes en Gamma.* PhD thesis, Université de Rennes 1, France, July 1993.