GAMMA AND THE CHEMICAL REACTION MODEL : TEN YEARS AFTER

J.-P. BANÂTRE, D. LE MÉTAYER
Irisa/Inria,
Campus de Beaulieu,
35042 Rennes, France
email: jpbanatre/lemetayer@irisa.fr

Gamma was originally proposed in 1986 as a formalism for the definition of programs without artificial sequentiality. The basic idea underlying the formalism is to describe computation as a form of *chemical reaction* on a collection of individual pieces of data. Due to the very minimal nature of the language, and its absence of sequential bias, it has been possible to exploit this initial paradigm in various directions. This paper reviews most of the work done by various groups along these lines and the current perspectives of our own research on Gamma. For the sake of clarity we separate the contributions in three categories: (1) the relevance of the chemical reaction model for software engineering, (2) extensions of the original model and (3) implementation issues.

1 The basic chemical reaction model

The notion of sequential computation has played a central rôle in the design of most programming languages in the past. This state of affairs was justified by at least two reasons:

- Sequential models of execution provide a good form of abstraction of algorithms, matching the intuitive perception of a program defined as a "recipe" for preparing the desired result.
- Actual implementations of programs were made on single processor architectures, reflecting this abstract sequential view.

However the computer science landscape has evolved considerably since then and the limitations of the sequential view have become more and more obvious. These changes have been caused by the tremendous increase in the size and the complexity of real software applications and the dramatic progress in the hardware technology. Let us examine these two issues in turn and assess their impact on programming languages.

1. **The evolution of software:** as a result of the growing needs for information processing and the decreasing cost of hardware, we have seen a tremendous proliferation of software systems. In particular embedded

systems affect basically every aspect of our daily life (remote control units for television sets, switching telephone networks, automatic train control, ...) [51]. This evolution introduces new problems:

- More and more software systems are used in safety critical applications (nuclear industry, airplane control, ...) and require a very high level of reliability. The only way to meet this requirement is to develop software in a rigorous way relying as much as possible on "formal methods". A formal language (like Z, VDM, or B) can be used to specify the software in a non ambiguous way and the specification should serve as a basis for the development of the software. Ideally, the software product should be proven correct with respect to the specification. This implies that a "discipline of programming" is crucially required, which places strong requirements on languages: the programming languages should be carefully designed in order to make it possible to prove properties of programs.
- Softwares tend to grow in size and complexity; they are often developed through a long period of time and become extremely difficult to understand and to maintain. The cost incurred by this complexity is becoming a serious concern and a major challenge today is to provide ways of organising software in order to make big applications manageable and to favour the reuse of existing products. Several languages have been proposed recently to tackle these problems: they are called software architecture languages [2,37], or coordination languages [18]. A key feature of these languages is to allow the description of interactions between individual components (which may sometimes be written in different programming languages, leading to a new class of challenging "inter-operability" problems).

Due to their sequential bias, most traditional programming languages are not well suited to take up these challenges: they often lead to programs with extremely complex correctness proofs and they do not provide the required level of abstraction to describe software architectures.

2. **The evolution of hardware:** we have seen in the last few years the widespread development of electronic networks made possible by the progress in communication technology. This trend is likely to be accelerated in the future. As a consequence, a computer can no longer be considered in isolation; it should rather be seen as a node in a graph representing a distributed system. Individual computers themselves are no longer single processors: parallelism is now integrated in various ways

and at all levels of the computation (from low-level pipelining and superscalar processors to shared-memory multiprocessors and fine-grained parallel machines). Programming such machines obviously requires parallel languages and models of computation.

The situation created by this double evolution has placed new needs on the design of languages: sequentiality should no longer be seen as the prime programming paradigm but just as one of the possible forms of cooperation between individual entities. The Gamma formalism was proposed ten years ago precisely to capture the intuition of computation as the global evolution of a collection of atomic values interacting freely. Gamma is a kernel language which 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 a pair (*Reaction condition*, *Action*). Execution proceeds by replacing in the multiset elements satisfying the reaction condition by the products of the action. The result is obtained when a stable state is reached, that is to say when no more reactions can take place. The following is an example of a Gamma program computing the maximum element of a non-empty set.

$$max\ :\ x,\ y \rightarrow\ y\ \Leftarrow x \leq y$$

$x \leq y$ specifies a property to be satisfied by the selected elements x and y. These elements are replaced in the set by the value y. Nothing is said in this definition about the order of evaluation of the comparisons. If several disjoint pairs of elements satisfy the condition, the reactions can be performed in parallel. Let us consider as another introductory example a sorting program. We represent a sequence as a set of pairs $(index, value)$ and the program exchanges ill-ordered values until a stable state is reached and all values are well-ordered.

$$sort\ :\ (i,x),\ (j,y) \rightarrow\ (i,y),\ (j,x)\ \Leftarrow (i > j)\ and\ (x < y)$$

The possibility of getting rid of artificial sequentiality in Gamma has two important consequences:

- It confers a very high level nature to the language and allows the programmer to describe programs in a very abstract way. In some sense, one can say that it is possible in Gamma to express the very "idea" of an algorithm without any unnecessary linguistic idiosyncrasy (like "exchange any ill-ordered values until all values are well ordered" for the sorting algorithm). This also makes Gamma suitable as an intermediate language in the program derivation process: Gamma programs are easier to prove correct with respect to a specification and they can be refined for the sake of efficiency in a second stage. This refinemement may involve the introduction of extra sequentiality but the crucial methodological advantage of the approach is that logical issues can be decoupled from implementation issues. Actually Gamma was originally proposed in the context of a work on systematic program derivation [9].

- Because Gamma programs do not have any sequential bias, the language naturally leads to the construction of parallel programs (in fact it is much harder to write a sequential program than a parallel program in Gamma). It also makes Gamma a potential candidate for a software architecture language in which atomic actions would be individual pieces of software using the multiset as the cooperation facility.

To conclude this introduction, let us quote E.W.Dijkstra twenty years ago [28]: *"Another lesson we should have learned from the recent past is that the development of "richer" or "more powerful" programming languages was a mistake in the sense that these baroque monstruosities, these conglomerations of idiosyncrasies, are really unmanageable, both mechanically and mentally. I see a great future for very systematic and very modest programming languages"*. We believe that this statement is more relevant than ever and the very minimal nature of the original Gamma formalism is one factor which has made possible the various developments that are sketched in this paper. For the sake of clarity, we provide in section 2 the basic intuitions about the programming style entailed by Gamma and its relevance for program construction and software engineering. In section 3 we describe a number of variations and elaborations on the original chemical reaction metaphor. Section 4 is devoted to implementation issues and section 5 is a sketch of related work and research perspectives in this area.

2 The chemical reaction as a basis for software engineering

Using the chemical reaction model as a basic paradigm can have a deep effect on our way of thinking about algorithms. We first try to convey the programming style favoured by Gamma through some very simple examples. Then we introduce five basic programming schemes, called "tropes" which have emerged from our experience in writing Gamma programs. Taking a slightly more formal view, we show that it is possible to extract useful properties from the definition of a Gamma program or, alternatively, to derive Gamma programs from specifications in a systematic way.

2.1 A new programming style

We first come back on the straightforward *max* program defined in the introduction to illustrate some distinguishing features of Gamma:

$$max \ : \ x, \ y \rightarrow \ y \ \Leftarrow x \leq y$$

In order to write a program computing the maximum of a set of values in a "traditional" language, we would first have to choose a representation for the set. This representation could typically be an array for an imperative language or a list for a declarative language. The program would be defined as an iteration through the array, or a recursive walk through the list. The important point is that the data structure would impose constraints on the order in which elements are accessed. Of course, parallel versions of imperative or functional programs can be defined (solutions based on the "divide and conquer" paradigm for example), but none of them can really model the total absence of ordering between elements that is achieved by the Gamma program. The essential feature of the Gamma programming style is that a data structure is no longer seen as a hierarchy that has to be walked through or decomposed by the program in order to extract atomic values. Atomic values are gathered into one single bag and the computation is the result of their individual interactions. A related notion is the "locality principle" in Gamma: individual values may react together and produce new values in a completely independent way. As a consequence, a reaction condition cannot include any global condition on the mutiset such as $\forall$-properties or properties on the cardinality of the multiset. The locality principle is crucial because it makes it easier to reason about programs and it encapsulates the intuition that there is no hidden control constraints in Gamma programs.

Let us now consider the problem of computing the prime numbers less than a given value n. The basic idea of the algorithm can be described as follows: "start with the set of values from 2 to n and remove from this set any element which is the multiple of another element". So the Gamma program is built as the sequential composition of *iota* which computes the set of values from 2 to n and *rem* which removes multiples. The program *iota* itself is made of two reactions: the first one splits an interval (x, y) with $x \neq y$ in two parts and the second one replaces any interval (x, x) by the value x.

$$primes(n) \ = \ rem(iota(\{(2, n)\}))$$

$$\begin{array}{lcl} iota & = & (x, y) \rightarrow (x, [(x+y)/2]) \ , \ ([(x+y)/2]+1, y) \Leftarrow x \neq y \\ & & (x, y) \rightarrow x \Leftarrow x = y \\ rem & = & x \ , \ y \rightarrow y \Leftarrow multiple(x, y) \end{array}$$

The first reaction increases the size of the multiset, the second one keeps it constant and the third one makes the multiset shrink. In contrast with the usual sequential or parallel solutions to this problem (usually based on the successive application of sieves [10]), the Gamma program proceeds through a collection of atomic actions applying on individual and independent pieces of data.

Another program exhibiting these expansion and shrinking phases is the Gamma version of the fibonacci function:

$$fib(n) \; = \; add(dec_1(n))$$

$$\begin{array}{rcl} dec_1 & = & x \rightarrow x-1 \; , \; x-2 \Leftarrow x > 1 \\ & & x \rightarrow 1 \Leftarrow x = 0 \\ add & = & x \; , \; y \rightarrow x + y \Leftarrow True \end{array}$$

The initial value is decomposed by dec_1 into a number of ones which are then summed up by add to produce the result. The first phase corresponds to the recursive descent in the usual functional definition

$$fib(x) \; = \; if \; x \; \leq \; 1 \; then \; 1 \; else \; fib(x-1) \; + \; fib(x-2)$$

while the reduction phase is the counterpart of the recursive ascent. However the Gamma program does not introduce any constraint on the way the additions are carried out, which contrasts with the functional version in which additions must be performed following the order imposed by the recursion tree of the execution.

As a last example of this introduction, let us consider the "maximum segment sum" problem. The input parameter is a sequence of integers. A segment is a subsequence of consecutive elements and the sum of a segment is the sum of its values. The program returns the maximum segment sum of the initial sequence. The elements of the multiset are triples (i, x, s) where i is the position of value x in the sequence and s is the maximum sum (computed so far) of segments ending at position i. The s field of each triple is originally set to the x field. The program max_l computes local maxima and max_g returns the global maximum.

$$maxss(M) \; = \; max_g(max_l(M))$$

$$\begin{array}{rcl} max_l & = & (i,x,s) \; , \; (i',x',s') \rightarrow (i,x,s) \; , \; (i',x',s+x') \\ & & \qquad \Leftarrow (i' = i+1) \; and \; (s + x' > s') \\ max_g & = & (i,x,s) \; , \; (i',x',s') \rightarrow (i',x',s') \Leftarrow s' > s \end{array}$$

2.2 The tropes: five basic programming schemes

The reader may have noticed a number a recurrent programming patterns in the small examples presented in the previous section. After some experience in writing Gamma programs, we came to the conclusion that a very small number of program schemes were indeed necessary to write most applications. We focus in this section on five of these schemes called *tropes* for:

Transmuter, **R**educer, **OP**timiser, **E**xpander, and **S**elector.

The tropes are schemata for basic reaction-action pairs, and are defined in the following way:

$$
\begin{array}{lcl}
\mathcal{T}(C,f) & = & x \rightarrow f(x) \Leftarrow C(x) \\
\mathcal{R}(C,f) & = & x,y \rightarrow f(x,y) \Leftarrow C(x,y) \\
\mathcal{O}(<,f_1,f_2,S) & = & x,y \rightarrow f_1(x,y), f_2(x,y) \Leftarrow (f_1(x,y), f_2(x,y)) < (x,y) \\
 & & \qquad\qquad \mathit{and}\ S(x,y)\ \mathit{and}\ S(f_1(x,y), f_2(x,y)) \\
\mathcal{E}(C,f_1,f_2) & = & x \rightarrow f_1(x), f_2(x) \Leftarrow C(x) \\
\mathcal{S}_{i,j}(C) & = & x_1,...,x_i \rightarrow x_j,...,x_i \Leftarrow C(x_1,...,x_i) \\
 & & \qquad\qquad (\mathit{where}\ 1 < j \leq (i+1))
\end{array}
$$

Notice that the reducer and the selector strictly decrease the size of the multiset; the expander increases its size; the transmuter and the optimiser keep its size constant. We first provide the intuition behind each of these tropes, then we present examples involving several tropes.

Transmuter: The transmuter applies the same operation to all the elements of the multiset until no element satisfies the condition C. For example the following program returns, for each initial triple $(n,m,0)$, a triple whose third component records the number of times m is a multiple of n.

$$
\begin{array}{rcl}
nt\ =\ \mathcal{T}(C,f)\ \textbf{where}\ C((n,m,k)) & = & \mathit{multiple}(m,n) \\
f((n,m,k)) & = & (n, m/n, k+1)
\end{array}
$$

The second reaction in the definition of *iota* in the previous section can also be expressed as a transmuter:

$$
\begin{array}{rcl}
rp\ =\ \mathcal{T}(C,f)\ \textbf{where}\ C((x,y)) & = & (x=y) \\
f((x,y)) & = & x
\end{array}
$$

Reducer: This trope reduces the size of the multiset by applying a function to pairs of elements satisfying a given condition. The counterpart of the traditional functional *reduce* operator can be obtained with an always true reaction condition. For instance the program *add* which appeared in the definition of *fib* can de expressed as:

$$add \;=\; \mathcal{R}(C, f) \textbf{ where } \begin{array}{rcl} C((x,y)) & = & True \\ f((x,y)) & = & x+y \end{array}$$

Expander: The expander is used to decompose the elements of a multiset into a collection of basic values. For example *ones* decomposes positive values n into n occurrences of 1s.

$$ones \;=\; \mathcal{E}(C, f_1, f_2) \textbf{ where } \begin{array}{rcl} C(x) & = & x > 1 \\ f_1(x) & = & x - 1 \quad f_2(x) = 1 \end{array}$$

The program *iota* can now be redefined as the composition of an expander and a transmuter:

$$\begin{array}{rcl} iota(M) & = & \mathcal{T}(C_1, f_1)\ (\ \mathcal{E}(C_2, f_2, f_3)\ (M)) \quad \textbf{where} \\ C_1((x,y)) & = & (x \;=\; y) \quad f_1((x,y)) = x \\ C_2((x,y)) & = & (x \;\neq\; y) \\ f_2((x,y)) & = & (x, [(x+y)/2]) \quad f_3((x,y)) = ([(x+y)/2]+1, y) \end{array}$$

Selector: The selector acts as a filter, removing from the multiset elements satisfying a certain condition. For example the program *max* can de defined as: $max \;=\; \mathcal{S}_{2,2}(\leq)$. The program *rem* used in the definition of *primes* is also an example of selector: $rem \;=\; \mathcal{S}_{2,2}(multiple)$. The program *primes* can now be expressed as follows:

$$primes(n) \;=\; rem\ (iota(\{(2,n)\}))$$

Note that the selector $\mathcal{S}_{i,i+1}(C)$ corresponds to a reaction which removes the i selected elements from the multiset.

Optimiser: $\mathcal{O}(<, f_1, f_2, S)$ optimises the multiset according to a particular criterion (expressed through the ordering $<$) while preserving the structure of the multiset (described by the relation S). The *sort* program presented in the first section is an instance of the optimiser scheme:

$$\begin{array}{rcl} sort \;=\; \mathcal{O}(\ll, f_1, f_2, S) \quad \textbf{where} & & \\ f_1((i,x),(j,y)) & = & (i,y) \\ f_2((i,x),(j,y)) & = & (j,x) \\ ((i,x),(j,y)) \ll ((i',x'),(j',y')) & \equiv & (y \;<\; y') \\ S((i,x),(j,y)) & = & (i \;>\; j) \end{array}$$

We end this section with two further examples involving combinations of tropes. Consider first the longest upsequence problem. A subsequence is obtained from a sequence by deleting some (not necessarily adjacent) values. A subsequence is called an upsequence if its values are in non-decreasing order. The problem is to compute the length of the longest upsequence of a sequence. We represent a sequence as a set of triples (n, x_n, l_n), where x_n is the value at index n and l_n is the length of the longest known upsequence ending at index n. The solution using tropes is the composition of an optimiser (for finding the length of the longest upsequence ending at each particular index) and a selector (for evaluating the maximum of these lengths).

$$
\begin{array}{lll}
lup(M) & = & maxl\ (\ li(M)) \\
\\
li & = & \mathcal{O}(\ll, f_1, f_2, S) \quad \textbf{where} \\
 & & f_1((n, x_n, l_n), (k, x_k, l_k)) \ = \ (n, x_n, l_n) \\
 & & f_2((n, x_n, l_n), (k, x_k, l_k)) \ = \ (k, x_k, l_n + 1) \\
 & & ((n, x_n, l_n), (k, x_k, l_k)) \ll \ ((n', x'_n, l'_n), (k', x'_k, l'_k)) \equiv (l_k > l'_k) \\
 & & S((n, x_n, l_n), (k, x_k, l_k)) = ((n \ < \ k) \ and \ (x_n \ \leq \ x_k)) \\
\\
maxl & = & \mathcal{S}_{2,2}(C) \quad \textbf{where} \\
 & & C((n, x_n, l_n), (k, x_k, l_k)) = l_n \leq l_k
\end{array}
$$

The maximum segment sum program presented in the previous section can be defined in terms of tropes in a very similar way. The fibonacci function can be expressed as the following combination of tropes:

$$
\begin{array}{lll}
fib(n) & = & add\ (\ zero\ (\ dec\ (\{n\}))) \\
dec & = & \mathcal{E}(C, f_1, f_2) \quad \textbf{where} \\
 & & C(x) \ = \ x \ > \ 1, \ \ f_1(x) \ = \ x \ - \ 1, \ \ f_2(x) \ = \ x \ - \ 2 \\
zero & = & \mathcal{T}(C, f) \quad \textbf{where} \\
 & & C(x) \ = \ (x \ = \ 0), \ \ f(x) \ = \ 1 \\
add & = & \mathcal{R}(C, f) \quad \textbf{where} \\
 & & C(x, y) \ = \ True\ , \ \ f(x, y) \ = \ x + y
\end{array}
$$

2.3 Further examples

The interested reader can find in [10] a longer series of examples chosen from a wider range of domains (string processing problems, graph problems, geometric problems). We just sketch in this section a small selection of applications that we consider more significant, either because of their size or because of their target domain.

Image processing applications

Gamma has been used in a project aiming at experimenting high-level programming languages for prototyping image processing applications [26]. The application was the recognition of the tridimensional topography of the vascular cerebral network from two radiographies. The treatment is decomposed into three main phases:

1. The first step consists in discovering the "skeleton" of each image by extracting a number of significant pixels.
2. The second step gathers consecutive pixels into segments of a given length.
3. The third step tries to establish correspondences between segments of the two images.

The first two steps aim at reducing the combinatorics of the problem and pave the way for the third step. The correspondence between segments can then be used to derive a correspondence between pixels and build the tridimensional representation.

A version of this application written in PL/1 was in use before the start of the experiment but it was getting huge and quite difficult to master. One of the benefits of rewriting the application in Gamma has been a better understanding of the key steps of the application and the discovery of a number of bugs in the original software. So Gamma has been used in this context as an executable specification language and it turned out to be very well suited to the description of this class of algorithms. The basic reason is probably that many treatments in image processing are naturally expressed as collections of local applications of specific rules.

In order to convey some intuition about the relevance of Gamma for this kind of applications, we describe a small part of the first stage called the edge detection problem. Each point of the image is originally associated with a grey intensity level. Then an intensity gradient is computed at each point and edges are defined as the points where the gradient is greater than a given threshold T. The gradient at a point is computed relative to its neighbours: only points at a distance d less than D are considered for the computation of the gradient. The gradient at a point P is defined in the following way:

$$\begin{array}{l} G(P) \;=\; maximum(neighbourhood) \;-\; minimum(neighbourhood) \\ \textbf{where}\;\; neighbourhood \;=\; \{intensity(P') \mid distance(P,P') < D\} \end{array}$$

We use a multiset of quadruples (P, l, min, max). P is the coordinates of the point, l is its intensity level and min and max are the current values of respectively $minimum(neighbourhood)$ and $maximum(neighbourhood)$. In the sequel, we use capital letters X, Y to denote quadruples and p, l, min, max to access the corresponding fields. The initial value of the min and max fields is l. The evaluation consists in decreasing min and increasing max until the limit values are reached. This is achieved by the program dci. The program sel discards the points where the gradient is less than the threshold and removes unnecessary fields from the remaining elements.

$$edges(M) \;=\; sel(dci(M))$$

$$\begin{array}{rcl}
dci & = & X\ ,\ Y \rightarrow (X.P, X.l, Y.l, X.max)\ ,\ Y \\
 & & \quad \Leftarrow (Y.l < X.min)\ and\ distance(X.P, Y.P)\ <\ D \\
 & & X\ ,\ Y \rightarrow (X.P, X.l, X.min, Y.l)\ ,\ Y \\
 & & \quad \Leftarrow (Y.l > X.max)\ and\ distance(X.P, Y.P)\ <\ D
\end{array}$$

$$\begin{array}{rcl}
sel & = & X \rightarrow \quad \Leftarrow X.max - X.min < T \\
 & & X \rightarrow (X.P, X.l) \Leftarrow X.max - X.min \geq T
\end{array}$$

A version of this program in terms of tropes is given in section 3.2. Another example of application of Gamma to image processing is reported in [44]. The aim of this application is to generate fractals to model the growth of biological objects. Again, the terseness and the facility of the expression of the problem is Gamma was seen as a great advantage. In both experiences however, the lack of efficient general purpose implementation of Gamma was mentioned as a serious drawback because it prevented any test on large examples. We postpone the discussion on implementation issues until section 4.

Reactive programming

The problems we have considered so far are essentially algorithmic. We suggest how Gamma can be used in a quite different way in this subsection: we are no longer interested in the result of the evaluation (the programs are typically non terminating) but rather in the possible values of the multiset during the computation. In [52], an operating system kernel is defined in Gamma (extended with a fairness assumption) and proven correct in a framework inspired by the Unity logic[17]. The basis of the solution is a *System* program which manipulates structures of the form

$$[P_i, S_i, M_i, C_i]$$

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 *System* itself. The *System* program includes rules of the form:

$$[P_i, S_i, M_i, C_i] \rightarrow \quad [P_i, S_i', M_i', C_i'] \quad \Leftarrow Ready(P_i, S_i)$$

$$\textbf{where} \;\; [S_i', M_i', C_i'] = P_i([S_i, M_i, C_i])$$

$$[P_i, S_i, M_i, C_i + \{(\textbf{in}, m)\}] \rightarrow [P_i, S_i, M_i + \{m\}, C_i] \Leftarrow True$$

$$[P_i, S_i, M_i, C_i + \{(\textbf{out}, j, m)\}] \ , \ [P_j, S_j, M_j, C_j] \rightarrow$$

$$[P_i, S_i, M_i, C_i] \ , \ [P_j, S_j, M_j, C_j + \{(\textbf{in}, m)\}] \Leftarrow True$$

The first reaction describes a sequential computation and the remaining two deal with communication between processes through their mailboxes.

An important result of the work described in [52] is the definition of a temporal logic for a version of Gamma extended with a fairness assumption and the derivation of the kernel of a file management system by successive refinements from a temporal logic specification. Each refinement step results in a greater level of detail in the definition of the network of processes. We are not aware of comparable attempts in the area of operating systems.

Software architecture

Another related area of application which attracts a growing interest is the formal definition of software architectures. As stated in [2], *"Software systems become more complex and the overall system structure - or software architecture - becomes a central design problem. An important step towards an ingineering discipline of software is a formal basis for describing and analysing these designs"*. Typical examples of software architectures are the "client-server organisation", "layered systems", "blackboard architecture". Despite the popularity of this topic, little attention has focused on methods for comparing software architectures or proving that they satisfy certain properties. One major reason which makes these tasks difficult is the lack of common and formally based language for describing software architectures. These descriptions are typically expressed informally with box and lines drawings indicating the global organisation of computational entities and the interactions between them [2].

The chemical reaction model has been used recently as a formal basis for specifying software architectures [37]. The application considered in [37] is a multiphase compiler and two architectures are defined in a variation of the basic

Gamma model called the "chemical abstract machine" [12,14] which is described in section 3.1. The different phases of the compiler are called *lexer, parser, semantor, optimiser and generator*. An initial phase called *text* generates the source text. The types of the data elements circulating in the architecture are *char, tok, phr, cophr, obj*. The elements of the multiset have one of the following forms:

$$i(t_1) \ \diamond\ o(t_2) \ \diamond\ phase$$

$$o(t_1) \ \diamond\ phase \ \diamond\ i(t_2)$$

$$phase \ \diamond\ i(t_1) \ \diamond\ o(t_2)$$

where $\diamond$ is a free constructor, t_1 and t_2 represent data types and *phase* is one of the phases mentioned above. An element starting with $i(t_1)$ (resp. $o(t_1)$) corresponds to a phase which is consuming inputs (resp. producing outputs). An element starting with *phase* is not ready to interact. So $i(t_1)$ and $o(t_1)$ can be seen as ports defining the communications which can take place in a given state.

The following is a typical reaction in the definition of an architecture:

$$i(d_1) \ \diamond\ o(d_2) \ \diamond\ m_1 \ ,\ o(d_1) \ \diamond\ m_2 \ \diamond\ i(d_3)$$

$$\rightarrow$$

$$o(d_2) \ \diamond\ m_1 \ \diamond\ i(d_1) \ ,\ m_2 \ \diamond\ i(d_3) \ \diamond\ o(d_1)$$

This rule describes pairwise communication between processing elements: m_1 consumes input d_1 produced as output by another processing element m_2. For example, the reaction:

$$i(tok) \ \diamond\ o(phr) \ \diamond\ parser \ ,\ o(tok) \ \diamond\ lexer \ \diamond\ i(char)$$

$$\rightarrow$$

$$o(phr) \ \diamond\ parser \ \diamond\ i(tok) \ ,\ lexer \ \diamond\ i(char) \ \diamond\ o(tok)$$

represents the consumption by the parser of tokens produced by the lexer. At the end of the reaction, the parser is ready to produce its output and the lexer is inert because it has completed its job. In fact another reaction may be applied later to make it active again to process another piece of text.

One major benefit of the approach is that it makes it possible to define several architectures for a given application and compare them in a formal way. As an example, [37] defines a correspondence between multisets generated by two versions of the multiphase compiler and establishes a form of bisimulation between the two architectures. They also prove normalisation properties of the programs.

The approach described in [31] focuses on the interconnection between individual components of the software architecture. It is based on a version of Gamma which makes it possible to manipulate relations between values of the multiset exactly in the same way as values themselves. This extension, called Structured Gamma, is presented in section 3.4. Its relevance for the description of software architecture comes from the fact that the connections between individual pieces of software become "first class" objects and invariance properties can be proven on the structure of the network. For example, the architecture of the multiphase compiler suggested above would be characterised by multisets of a particular type including values corresponding to the processing elements (as above) and relations corresponding to the links between these values (this rôle is played implicitly by the data types in the communication ports in [37]). The evolution of the computation is again defined in terms of reaction rules but it is possible to prove that these rules maintain the invariant defining the architecture (this implies, for instance, that the parser can only consume values produced by the lexer). In other words, the definition of the static aspect of the software architecture is separated from the specification of its dynamic component. The former is the type structure of the multisets manipulated by the program defining the latter.

2.4 Program derivation

In the previous sections, we tried to convey the programming style entailed by the chemical reaction model through a series of examples. We have not said a word yet on the possibility of formal reasoning on Gamma programs. A semantics of the language in terms of multiset rewriting was proposed and discussed in [9]. We do not intend to go into the formal aspects in this paper but we sould like to suggest the techniques that can be used to prove properties of programs or to derive programs from specifications.

In order to prove the correctness of a program in an imperative language, a common practice consists in splitting the property into two parts: the *invariant* which holds during the whole computation, and the *variant* which is required to hold only at the end of the computation. In the case of total correctness, it is also necessary to prove that the program must terminate. The important observation concerning the variant property is that a Gamma program terminates when no more reaction can take place, which means that no tuples of elements satisfy the reaction condition. So the obtain the variant of the program by taking the negation of the reaction condition. Let us consider as an example the *sort* program introduced in section 1. The reaction condition corresponds to the property:

$$\exists (i, x) \in M.\ \exists (j, y) \in M.\ (i > j)\ and\ (x < y)$$

and its negation

$$\forall (i, x) \in M.\ \forall (j, y) \in M.\ (i > j) \Rightarrow x \geq y$$

This variant is very informative indeed since it is the well-ordering property. The invariant of the program must ensure that the set of indexes and the multiset of values are constant. This can be checked by a simple inspection of the action

$$A((i, x), (j, y)) \;=\; \{(i, y),\ (j, x)\}$$

It is easy to see that the global invariance follows from the local invariance. In order to prove the termination of the program, we have to provide a well-founded ordering (an ordering such that there is no infinite descending sequences of elements) and to show that the application of an action decreases the multiset according to this ordering. To this aim, we can resort to a result from [27] allowing the derivation of a well-founded ordering on multisets from a well-founded ordering on elements of the multiset. Let $\succ$ be an ordering on V and $\gg$ be the ordering on $Multisets(V)$ defined in the following way:

$$M \gg M' \Leftrightarrow$$

$$\exists X, Y \in Multisets(V).\ X \neq \emptyset \ \ and$$

$$X \subseteq M \ \ and \ \ M' = (M - X) + Y \ \ and \ \ (\forall y \in Y.\ \exists x \in X.\ x \succ y)$$

The ordering $\gg$ on $Multisets(V)$ is well-founded if and only if the ordering $\succ$ on V is well-founded. This result is fortunate because the definition of $\gg$ precisely mimicks the behaviour of Gamma (removing elements from the multiset and inserting new elements). The significance of this result is that it allows us to reduce the proof of termination, which is essentially a global property, to a local condition. In order to prove the termination of the *sort* program, we can use the following ordering on the elements of the multiset:

$$(i, x) \sqsubseteq (i', x') \Leftrightarrow (i \geq i' \ \ and \ \ x' \geq x)$$

It is easy to see that this ordering is well-founded (the set of indexes and the multiset of values are finite), so the corresponding multiset ordering is also well-founded. We are left with the proof that for each value produced by the action, we can find a consumed value which is strictly greater. To prove this we observe that:

$$(i, y) \sqsubset (j, y) \quad and \quad (j, x) \sqsubset (j, y)$$

This concludes the correctness proof of the sort program.

Rather than proving a program *a posteriori*, it may be more appropriate to start from a specification and try to construct the program in a systematic way. The derived program is then correct by construction. A method for the derivation of Gamma programs from specifications in first order logic is proposed in [9]. The basic strategy consists in splitting the specification into a conjonction of two properties which will play the rôles of the invariant and the variant of the program to be derived. The invariant is chosen as the part of the specification which is satisfied by the input multiset (or that can be established by an initialisation program). If the variant involves only $\forall$ quantifiers than its negation yields the reaction condition of the program directly. The technique for deriving the action consists in validating the variant locally while maintaining the invariant. These two constraints are very often strong enough to guide the construction of the action. Let us consider as an example the *rem* program in the definition of *prime* in section 2.1. The input multiset is $\{2, \ldots, n\}$ and a possible specification of the result M is the following:

$$M \subseteq \{2, \ldots, n\} \tag{1}$$
$$\forall x \in \{2, \ldots, n\}.\ (\forall y \in \{2, \ldots, n\}.\ \neg multiple(x, y)) \Rightarrow x \in M \tag{2}$$
$$\forall x, y \in M.\ \neg multiple(x, y) \tag{3}$$

Both properties (1) and (2) are satisfied by the input multiset $\{2, \ldots, n\}$, so the invariant is defined as $I = (1) \wedge (2)$ and the variant is $V = (3)$. The negation of the variant is

$$\exists x, y \in M.\ multiple(x, y)$$

which yields to the reaction condition $multiple(x, y)$. The action must satisfy the invariant which means that no value outside $\{2, \ldots, n\}$ can be added to the multiset and no value should be removed from the multiset unless it is the multiple of another value. On the other hand, the action should establish the variant locally which means that the returned values should not contain any multiples. So the action cannot return both x and y and it cannot remove y: the only possibility is to return y; this action satisfies all the conditions and the derived program is:

$$rem \quad = \quad x\ ,\ y \rightarrow y \Leftarrow multiple(x, y)$$

The interested reader can find a more complete treatement of several examples in [9]. A slightly different approach is taken in [48] which introduces a very general form of specification and the derivation of the corresponding Gamma program. It is then shown that a number of classical and apparently unrelated problems (the knapsack, the shortests paths, the maximum segment sum and the longest upsequences problems) turn out to be instances of the generic specification. The generic derivation can then be instantiated to these applications, yielding the corresponding Gamma programs.

To conclude this section, let us stress the pervasive influence of the locality principle stated in the introduction in the correctness proofs and the derivations. Each part of the correctness proof of the *sort* program sketched above exploits this feature by reducing the global reasoning (manipulation of properties of the whole multiset) to a local reasoning (on the elements involved in a single reaction). One major benefit of the systematic derivation approach is that it sometimes leads to unexpected and parallel solutions for problems which are considered as inherently sequential. This is probably the best way to explore this new style of programming and to avoid the pitfall of trying to recast into the Gamma framework our traditional programming patterns.

3 Elaborations on the basic chemical reaction model

The chemical reaction model has served as the basis of a number of works which have led to several extensions of the minimal version described so far. In this section, we review the most important elaborations on the chemical reaction model that have been proposed in the literature. Some of these proposals have been put forward with theoretical objectives in mind, others aim at making the model more practically usable. Most of these extensions are quite significant and open new areas for research but it is important to note that none of them jeopardise the fundamental characteristics of the model which is the expression of computation as "the global result of the successive applications of local, independent, atomic reactions".

3.1 The chemical abstract machine

The chemical abstract machine (or *cham*) was proposed by Berry and Boudol [12] to describe the operational semantics of process calculi. The most important additions to Gamma are the notions of *membrane* and *airlock mechanism*. Membranes are used to encapsulate solutions and to force reactions to occur locally. In terms of multisets, a membrane can be used to introduce multiset of molecules inside a multiset that is to say *"to transform a solution into a* *single molecule"*[14]. The airlock mechanism is used to describe communications between an encapsulated solution and its environment. The reversible airlock operator $\triangleleft$ extracts a element m of a solution $\{m, m_1, \ldots, m_n\}$:

$$\{m, m_1, \ldots, m_n\} \rightleftharpoons \{m \triangleleft \{m_1, \ldots, m_n\}\}$$

The new molecule can react as a whole while the subsolution $\{m_1, \ldots, m_n\}$ is allowed to continue its internal reactions. So the main rôle of the airlock is to allow one molecule to be visible from outside the membrane and thus to take part in a reaction in the embedding solution. The need for membranes and airlocks emerged from the description of CCS [46] in Cham and especially the treatment of the restriction operation (which restricts the communication capabilities of a process to labels different from a particular value a). The computation rules of the cham are classified into general laws and two classes of rules:

- The general laws include the chemical law and the membrane law:

$$\frac{S \rightarrow S'}{S + S" \rightarrow S' + S"}$$

$$\frac{S \rightarrow S'}{\{C[S]\} \rightarrow \{C[S']\}}$$

  The former shows that reactions can be performed freely within any solution, which captures the locality principle. The latter allows reactions to take place within a membrane ($C[S]$ denotes any context of a solution S).

- The first class of rules corresponds to the proper reaction rules similar to the rules presented so far in the paper. The definition of a specific cham requires the specification of a syntax for molecules and the associated reaction rules. As an example, molecules can be CCS processes and the rule corresponding to communication in CCS would be:

$$\alpha.P\ ,\ \overline{\alpha}.Q \mapsto P\ ,\ Q$$

- The second kind of rules are called structural and they are reversible. They can be decomposed into two inverse relations $\rightharpoonup$ and $\rightharpoondown$ called respectively heating and cooling rules. The first ones break complex molecules into smaller ones, preparing them for future reactions, and the second ones rebuild heavy molecules from light ones. Continuing the CCS example, we have the structural rule:

$$(P \mid Q) \rightleftharpoons P\ ,\ Q$$

where | is the CCS parallel composition operator.

The cham was used in [12] to define the semantics of various process calculi (TCCS, Milner's π-calculus of mobile processes) and a concurrent lambda calculus. A cham for the call-by-need reduction strategy of λ-calculus is defined in [14]. The cham has inspired a number of other contributions. Let us mention some of them [14]:

- [1] uses a *linear cham* to describe the operational semantics of proof expressions for the classical linear logic.
- [47] defines an operational semantics of the π-calculus in a cham style.
- [38] describes a graph reduction in terms of a cham.
- [41] applies the cham in the context of the Facile implementation.

The cham approach illustrates the significance of multisets and their connection with concurrency. The fact that multisets are inherently unordered makes them suitable as a basis for modelling concurrency which is an essentially associative and commutative notion. As stated in [12]: *"In the SOS style of semantics, labelled transitions are necessary to overcome the rigidity of syntax when performing communications between two syntactically distant agents. ... On the contrary, in the cham, we just make the syntactic distance vanish by putting molecules into contact when they want to communicate, and their communication is direct."* As a consequence, this makes it possible to bring the semantics of concurrent systems nearer to the execution process of sequential languages, or the evaluation mechanism of functional languages [14].

3.2 Composition operators for Gamma

The Gamma programs that we have presented so far are made from a single block of reaction rules. The basic version of the language does not provide any facility for building complex programs from simple ones. For the sake of modularity, it is desirable that a language offers a rich set of operators for combining programs. It is also fundamental that these operators enjoy a useful collection of algebraic laws in order to make it possible to reason about programs. In this section, we sketch several proposals which have been made to extend Gamma with facilities for structuring programs.

[35] presents of a set of operators for Gamma and studies their semantics and the corresponding calculus of programs. The two basic operators considered in this paper are the sequential composition $P_1 \circ P_2$ and the parallel composition $P_1 + P_2$. The intuition behind $P_1 \circ P_2$ is that the stable multiset reached after

the execution of P_2 is given as argument to P_1. On the other hand, the result of $P_1 + P_2$ is obtained (roughly speaking) by executing the reactions of P_1 and P_2 (in any order, possibly in parallel), terminating only when neither can proceed further. The termination condition is particularly significant and heavily influences the choice of semantics for parallel composition. As an example of sequential composition of Gamma programs, let us consider another version of sort.

$$\begin{array}{ll} sort' : match \circ init & \\ \text{where} \quad init: & (x \rightarrow (1, x) \Leftarrow integer(x)) \\ \qquad match: & ((i, x), (j, y) \rightarrow (i, x), (i+1, y) \Leftarrow (x \leq y \ and \ i = j)) \end{array}$$

The program $sort'$ takes a multiset of integers and returns an increasing list encoded as a multiset of pairs $(index, value)$. The reaction $init$ gives each integer an initial rank of one. When this has been completed, $match$ takes any two elements of the same rank and increases the rank of the larger.

The case for parallel composition is slightly more involved. In fact $sort'$ could have been defined as well as:

$$sort' : match + init$$

because the reactions of $match$ can be executed in parallel with the reactions of $init$ (provided they apply on disjoined subsets, but this is implied by the fact that their respective reaction conditions are exclusive). As far as the semantics of parallel composition is concerned, the key point is that a *synchronised termination* of P_1 and P_2 is required for $P_1 + P_2$ to terminate. It may be the case that, at some stage of the computation, none of the reaction conditions of, P_1 (resp. P_2) holds; but some reactions by P_2 (resp. P_1) may create new values which will then be able to take part in reactions by P_1 (resp. P_2). This situation precisely occurs in the above example where no reaction of $match$ can take place in the initial multiset; but $init$ transforms the multiset and triggers subsequent reactions by $match$. Thus the termination condition of $P_1 + P_2$ indicates that neither P_1 nor P_2 can terminate unless both terminate and the composition as well. This contrasts with the *asynchronous termination* condition of most process calculi (where if P_1 terminates (reduces to *nil* then $P_1 \mid\mid P_2 \rightarrow P_2$). Gamma programs should rather be compared with rewriting systems, and their parallel composition with the union of rewriting systems. In this context, it is natural to say that a normal form is reached only when none of the systems possess a rule which can apply to the term.

This new vision of parallel composition and its combination with the sequential composition creates interesting semantical problems which are studied in [35]. [35] defines a set of program refinement and equivalence laws for parallel and sequential composition, by considering the input-output behaviour induced by an operational semantics. Particular attention is paid on conditions under which $P_1 \circ P_2$ can be transformed into $P_1 + P_2$ and vice-versa. These transformations are useful to improve the efficiency of a program with respect to some particular machine and implementation strategy. Let us take another example to illustrate this point:

$$
\begin{array}{l}
connected \quad = \quad singleton \circ (P_1 + P_2) \\
\quad \textbf{where} \\
P_1 : v, w, (m, n) \rightarrow v \cup w \Leftarrow nodes(v) \ and \ nodes(w) \ and \ m \in v \ and \ n \in w \\
P_2 : v, (m, n) \rightarrow v \Leftarrow nodes(v) \ and \ m \in v \ and \ n \in v
\end{array}
$$

This program is used to detect whether a graph is strongly connected or not. The initial multiset representation of the graph consists of the collection of singleton sets of *nodes*, together with the collection of *edges*. A pair (m, n) is used to represents an edge linking nodes m and n. It proceeds by building bigger and bigger aggregates of connected nodes (through P_1). (The predicate "nodes" simply allows the reactions to distinguish between an edge and a node set.) P_2 is used to remove edges connecting two nodes belonging to the same set. Once this process has stabilised, the graph is connected if all the nodes have been gathered into a single set. This is tested via the primitive *singleton* – not specified here.

Using the algebra of programs defined in [35] one can show, for example, that $P_1 + P_2$ is equivalent to $P_2 \circ P_1$ which means that all the reactions of P_2 can be postponed until no more P_1 reactions can take place. If the target architecture is a sequential one (or even a parallel one with relatively few processors), $P_2 \circ P_1$ will be more efficient because many useless tests of the reaction condition of P_2 will be avoided; however $P_1 + P_2$ might turn out to be a better version if executed on a massively parallel machine because unnecessary edges can be removed by P_2 at the same time as aggregates are built by P_1.

The operational semantics mentioned above is not compositional, which makes it difficult to use for reasoning about large programs. [53] proposes a solution to this problem by defining a compositional semantics of the language based on transition traces and presents a number of basic laws of programs in this semantics. Not all the laws established in the operational semantics remain valid in the transition trace semantics; this is because transition traces distinguish programs with identical input/output behaviour but which behave differently in different contexts. It is shown however that most interesting properties still hold, the great advantage of this new semantics being that these laws can be used in a modular way to prove properties of large programs. The results can be specialised to derive useful properties of tropes, and so, useful properties of common forms of Gamma programs.

As an application, let us consider the image processing program *edges* introduced in section 2.3.1. This program can be expressed in terms of tropes as follows:

$$edges \;=\; (disc \;+\; rf) \;\circ\; (decmin \;+\; incmax)$$

$$\begin{array}{rcl}
decmin & = & \mathcal{O}(\ll, f_1, f_2, S) \quad \textbf{where} \\
 & & \begin{array}{rcl} f_1(X,Y) & = & (X.P, X.l, Y.l, X.max) \\ f_2(X,Y) & = & Y \\ (X', Y') \ll (X,Y) & \equiv & (X'.min \;<\; X.min) \\ S(X,Y) & = & distance(X.P, Y.P) \;<\; D \end{array} \\
incmax & = & \mathcal{O}(\prec, f_1, f_2, S) \quad \textbf{where} \\
 & & \begin{array}{rcl} f_1(X,Y) & = & (X.P, X.l, X.min, Y.l) \\ f_2(X,Y) & = & Y \\ (X', Y') \prec (X,Y) & \equiv & (X'.max \;>\; X.max) \\ S(X,Y) & = & distance(X.P, Y.P) \;<\; D \end{array} \\
disc & = & \mathcal{S}_{1,2}(C) \quad \textbf{where} \\
 & & C(X) = (X.max \;-\; X.min) \;<\; T \\
rf & = & \mathcal{T}(C, f) \quad \textbf{where} \\
 & & \begin{array}{rcl} C(X) & = & (X.max \;-\; X.min) \;\geq\; T \\ f(X) & = & (X.P, X.l) \end{array}
\end{array}$$

The following transformations can be justified using the general properties of tropes derived from the compositional semantics:

$$\begin{array}{l}
disc \;+\; rf \;\Longrightarrow\; disc \;\circ\; rf \\
disc \;+\; rf \;\Longrightarrow\; rf \;\circ\; disc \\
decmin \;+\; incmax \;\Longrightarrow\; decmin \;\circ\; incmax \\
decmin \;+\; incmax \;\Longrightarrow\; incmax \;\circ\; decmin
\end{array}$$

Composition operators have been studied in a more general framework called *reduction systems* [54] which are sets equipped with some collection of binary rewrite relations. This work has led to a new graph representation of Gamma programs which forms a better basis for the study of compositional semantics and refinement laws. Roughly speaking, the nodes of the graphs correspond to the sets of basic programs (reaction conditions) which are active at a given stage of the computation and an edge represents an internal termination step. For example, the graph associated with the program

$$(P_1 \circ P_3) \;+\; (P_2 \circ P_4)$$

is defined by the nodes $N_1 = \{P_3, P_4\}$, $N_2 = \{P_1, P_4\}$, $N_3 = \{P_3, P_2\}$ and $N_4 = \{P_1, P_2\}$ and the edges (N_1, N_2), (N_1, N_3), (N_2, N_4) and (N_3, N_4). Control starts at node N_1 which is the root of the graph and can proceed either through N_2 (when P_3 terminates and P_1 can start) or through N_3 (if P_4 terminates before P_3). The final state is N_4. This representation seems to be well suited to the study of program logics and it has led to the discovery of additional laws of Gamma programs.

A different approach to the definition of a compositional semantics is taken in [22] which proposes an alternative definition for the parallel composition operator. The sequential composition is not context sensitive with respect to the parallel composition in [35]. For example, $(P \circ Q) \mid R$ can reduce to $P \mid R$ when the multiset becomes inert for the program Q. But R could possibly produce new values which could then be consumed by Q. [22] convincingly argues that this choice can lead to undesired computations and is not coherent with the choice of synchronous termination (which corresponds to a context sensitive condition): for $P + R$ to terminate, the multiset must be inert for both P and R. Their solution is based on a separation of reduction rules into proper transformations (which correspond to individual chemical reactions) and unproper transformations which modify the program but have no effect on the multiset. The resulting definition of the parallel operator restricts its non determinism and makes it possible to avoid the undesired computations. An observational equivalence based on the concept of strong bisimulation is defined and shown to be a congruence [22]. Other semantics of Gamma have been proposed, including [29] which defines a congruence based on transition assertions and [50] which describes the behaviour of Gamma programs using Lamport's Temporal Logic of Actions. A denotational semantics expressed in terms of transition traces is used in [32] to derive a program logic for Gamma. The derivation is based on Abramsky's framework of domain theory in logical form.

3.3 Higher-order Gamma

We have suggested in the previous section several proposals for the extension of Gamma with buit-in primitive operators. Another approach for introduction of composition operators in a language consists in providing a way for the programmer to define them as higher-order programs. This is the traditional view in the functional programming area and it requires to be able to manipulate programs as ordinary data. This is the approach followed in [40] which proposes a higher-order version of Gamma. The definition of Gamma used so far involves two different kinds of terms: the programs and the multisets. The multiset is the only data structure and programs are described as collections of pairs *(Reaction Condition, Action)*. The main extension of higher-order Gamma consists in unifying these two categories of expressions into a single notion of configuration. One important consequence of this approach is that active configurations may now occur inside multisets and reactions can take place (simultaneously) at different levels. Thus two conditions must be satisfied for a simple program to terminate: no tuple of elements satisfies the reaction condition and the multiset does not contain active elements.

A configuration is denoted:

$$[Prog, Var_1 = Multexp_1, \ldots, Var_n = Multexp_n].$$

It consists of a (possibly empty) program $Prog$ and a record of named multisets Var_i. A configuration with an empty program component is called passive, otherwise it is active. The record component of the configuration can be seen as the environment of the program. Each component of the environment is a typed multiset. Simple programs extract elements from these multisets and produce new elements. A stable component $Multexp_i$ of a configuration C can be obtained as the result of $C.Var_i$.

The operational semantics is essentially extended with the following rules to capture the higher-order features:

$$\frac{X \rightarrow X'}{\{X\} \oplus M \rightarrow \{X'\} \oplus M}$$

$$\frac{M_k \rightarrow M_k'}{[P, \ldots Var_k = M_k, \ldots] \rightarrow [P, \ldots Var_k = M_k', \ldots]}$$

The first and the second rule respectively account for the computation of active configurations inside multisets and for the transformation of multisets containing active configurations inside a configuration. Note that these rules are very similar to the chemical law and the membrane law of the cham (section 3.1).

Let us take one example to illustrate the expressive power provided by this extension. The application of the sequential composition operator to simple programs can be defined in higher-order Gamma, and thus does not need to be included as a primitive. $(P_2 \circ P_1)(M_0)$ is defined by the following configuration:

$$\begin{array}{l}[Q, E_1 = \{[P_1, M = M_0]\}, E_2 = \emptyset].E_2 \\ \textbf{where} \;\; Q = [\emptyset, M = M_1] : E_1 \rightarrow [P_2, M = M_1] : E_2\end{array}$$

E_1 is a multiset containing the active configuration $[P_1, M = M_0]$ initially. Note that Q reactions only apply to passive values of E_1 which means that M_1 must be a stable state for P_1. Then the new active configuration $[P_2, M = M_1]$

is inserted into E_2 and the computation of P_2 can start. When a stable state is obtained, it is extracted from the top-level configuration through the access operation denoted by $.E_2$.

Higher-order Gamma has been used to provide a formal specification of a "real life" distributed system [15]: the Lotus Notes groupware product. The relevance of the formalism is illustrated by the detection of errors in an initial specification, the elimination of overly restrictive data structures and the exposition of the natural parallelism of the replication algorithm. [40] shows how other useful combining forms can be defined in higher-order Gamma (including the chemical abstract machine). It is also possible to express more sophisticated control strategies such as the *scan vector model* suitable for execution on fine-grained parallel machines. Another generalisation of the chemical model to higher-order is presented in [23].

3.4 Structured Gamma

Sections 2 and 3 have illustrated the fact that the choice of the multiset as the unique data constructor is central in the design of Gamma. However, this may lead to programs which are unnecessary complex when the programmer needs to encode specific data structures. For example, it was necessary to resort to pairs *(index, value)* to represent sequences in the *sort* program. Trees or graphs can be encoded in a similar way. This lack of structuring is detrimental both for reasoning about programs and for implementing them. The proposal made in [31] is an attempt to solve this problem without jeopardising the basic qualities of the language. Let us point out in particular that it would not be acceptable to take the usual view of recursive type definitions because this would lead to a recursive style of programming and ruine the fundamental locality principle (because the data structure would then be manipulated as a whole).

The solution proposed in [31] is based on a notion of *structured multiset* which can be seen as a set of addresses satisfying specific relations and associated with a value. A type is defined in terms of rewrite rules and a structured multiset belongs to a type T if its underlying set of addresses satisfies the invariant expressed by the rewrite system defining T. As an example, the list type can be defined by the following rewrite system:

$$\begin{array}{lcl} List & \leftarrow & L\ x \\ L\ x & \leftarrow & S\ x\ y\ ,\ L\ y \\ L\ x & \leftarrow & T\ x \end{array}$$

which means that any multiset which can be reduced to the singleton *List* belongs to the list type. The variables in the rules are instantiated with addresses in the multiset. $L\ x$ can be seen as a non-terminal standing for a list starting at address x and $T\ x$ is a one element list. A circular list can be defined as follows:

$$\begin{array}{lcl} Circular & \leftarrow & L\ x\ x \\ L\ x\ y & \leftarrow & S\ x\ y \\ L\ x\ y & \leftarrow & L\ x\ z\ ,\ L\ z\ y \end{array}$$

Note that the use of different variable names in a rule is significant: two variables are instantiated with the same address if and only if the variables have the same name. In this definition, $L\ x\ y$ is the non terminal for a list starting at position x and ending at position y.

A reaction in Structured Gamma can:

- test and modify the relations on addresses.
- test and modify the values associated with the addresses.

Here are some examples of programs operating on lists:

$$\begin{array}{lcll} Sort : List & = & S\ a\ b\ ,\ \overline{a} < \overline{b} & \Rightarrow \\ & & S\ a\ b\ , & \\ & & a\ :=\ \overline{b}, & \\ & & b\ :=\ \overline{a} & \end{array}$$

$$\begin{array}{lcll} Mult : List & = & S\ a\ b\ ,\ S\ b\ c & \Rightarrow \\ & & S\ a\ c\ , & \\ & & a\ :=\ \overline{a} * \overline{b} & \end{array}$$

$$\begin{array}{lcll} Iota : List & = & T\ a\ ,\ \overline{a} > 1 & \Rightarrow \\ & & S\ a\ b\ ,\ T\ b\ , & \\ & & b\ :=\ \overline{a} - 1 & \end{array}$$

Actions are now described as assignments to given addresses. A consumed address which does not occur in the result of the action disappears from the multiset: this is the case for b in the $Mult$ program. On the other hand, new addresses can be added to the multiset with their value, like b in the $Iota$ program. Actions must also state explicitly how the relations are modified. For instance, the $Sort$ does not modify the S relation, but $Mult$ shrinks the list by removing the intermediate element b.

The significance of the approach is that the programmer can define his own types and programs can be checked according to the type definitions.

This verification can be made automatically using term rewriting techniques (based on the notion of critical pairs). For example it is possible to check that the three programs above manipulate multisets of type list: in other words, the list property is an invariant of the programs. It is important to notice that this new structuring possibility is obtained without sacrificing the fundamental qualities of the language. Gamma programs are just particular cases of Structured Gamma programs and Structured Gamma programs can be translated in a straightforward way into Gamma[31].

3.5 Gamma and logic programming

In its original setting, the two main features of Gamma are the multiset and a fixed collection of *(condition,action)* pairs. The types of the objects in the multiset and the language for the definition of the actions is left unspecified. As a consequence, Gamma can as well be seen as a *coordination language* (or a language for coordinating applications written in different languages). In this paper, we have defined actions as functions but proposals have been made recently for the integration of Gamma and logic programming languages. Two main questions have to be answered with respect to this integration:

- What kind of logical objects are contained in the multiset?
- How are the *(condition,action)* pairs defined?

A first approach, followed in [21], uses multisets of terms and describes conditions and actions as predicates. The semantics of an extension of logic programming with multisets is defined in [21] and a form of soundness and completeness are proven. It involves a definition of multiset unification and a careful integration with the operational semantics of Gamma, in which the "choices" made by reaction conditions are not backtrackable. This model is implemented as an extension of Gödel, a strongly typed logic programming language with a rich module system. This extension, called Gammalög, includes the sequential and parallel composition operators introduced in [35].

Another approach is followed in [55] where the objects in the multisets are goal formulas and the *(condition,action)* pairs are goal-directed deduction rules. This results in λLO, an extension of LO which can itself be seen as an elaboration on the basic chemical reaction model. LO [3,4,5] is based on a fragment of linear logic which admits goal-directed proofs. The notion of multisets of goal formulas is natural in LO because it materialises the difference between intuitionistic logic and linear logic. LO offers naturally multiple multisets because it permits to spawn subproofs associated with their own multisets. λLO can be seen as a higher-order extension of LO in the same way as λProlog is a higher-order extension of Prolog. Implication in goals provides the ability to construct (or augment) the program at run-time and the use of multisets leads to a uniform treatment of programs and data.

4 From chemistry to computing: implementation issues

As mentioned in previous sections, the philosophy of Gamma is to introduce a clear separation between correctness issues and efficiency issues in program design. Traditional programming languages encourage the emergence of solutions which are overconstrained when compared with the logic of the problem to be solved. In particular, sequential data structures (arrays, lists, ...) favour solutions based on iteration or recursion which are inherently sequential and limit dramatically the possibility of concurrent execution. In contrast, Gamma can be seen as a specification language which does not introduce unnecessary sequentiality. As a consequence, a Gamma program is usually far away from a real architecture and designing a reasonably efficient implementation of the language is not straightforward. This section reviews the efforts which have been made in order to improve the implementation of Gamma.

4.1 A sequential implementation of Gamma

Consider a very simple form of Gamma program:

$$x_1, \ldots, x_n \rightarrow f(x_1, \ldots, x_n) \Leftarrow R(x_1, \ldots, x_n)$$

A straightforward implementation of this program can be described by the following imperative program:

While tuples remain to be processed
  do
    choose a tuple $(x_1, \ldots, x_n)$ not yet processed;
    if $R(x_1, \ldots, x_n)$ **then**
      (1) **remove** $x_1, \ldots, x_n$ **from** M
      (2) **replace** them by $f(x_1, \ldots, x_n)$
  end

This very naïve implementation puts forward most of the problems which have to be tackled in order to produce a Gamma implementation with a realistic complexity. The hardest problem concerns the construction of all tuples to be checked for reaction. A blind approach to this problem leads to an untractable complexity but a thorough analysis of the possible relationships between the elements of the multiset and the shape of the reaction condition may lead to improvements which highly optimise the execution and produce acceptable performances. In his thesis, C. Creveuil [24] studied several optimisations which are summarised here.

One important source of inefficiency comes from useless (redundant or deemed to fail) checks of the reaction condition. Three optimisations can dramatically reduce the overhead resulting from this redundancy:

1. **Decomposition of the reaction condition**: instead of considering R as a whole, one may decompose it as a conjunction of simpler conditions like:

$$R(x_1, \ldots, x_n) = R_1(x_1) \wedge R_2(x_1, x_2) \wedge \ldots \wedge R_n(x_1, x_2, \ldots, x_n)$$

   The test for condition R is done incrementally, avoiding the construction of tuples whose prefix does not satisfy one R_i.

2. **Detection of neighbourhood relationships**: the analysis of the reaction condition may provide information which can be used to limit the search space. For example, it may happen from the reaction condition that only adjacent values can react, or that only values possessing a common "flag" can be confronted. These properties can be detected at compile time and, in some situations (some sorting examples, pattern detection in image processing applications), the run-time improvement is considerable.

3. **Control of the non-determinism:** The Gamma paradigm imposes no constraint on the way tuples are formed. [24] shows that limiting non-determinism by imposing an ordering in the choice of values to be checked against the reaction condition can be very fruitful.

An interesting conclusion of the work described in [24] is that well-known efficient versions of sequential algorithms (shortest path for instance) can be "rediscovered" and justified as the result of several opimisations of a naïve implementation of Gamma. It is very often the case that the most drastic optimisations rely on structural properties of the values belonging to the multiset (neighbourhood relationship, ordering in the choice of values ...).

Such properties are difficult to find automatically and the optimisations described above can be seen as further refinement steps rather than compilation techniques. This is the reason why several proposals have been made to enrich Gamma with features which could be exploited by a compiler to reduce the overhead associated with the "magic stirring" process [19,20,31,45]. We come back to these extensions in the conclusion.

4.2 *Parallel implementations of Gamma*

A property of Gamma which is often presented as an advantage is its potential for concurrent interpretation. In principle, due to the locality property, each tuple of elements fulfilling the reaction condition can be handled simultaneously. It should be clear however that managing all this parallelism efficiently can be a difficult task and complex choices have to be made in order to map the chemical model on parallel architectures. This section sketches some attempts to provide parallel implementations for Gamma programs.

Implementations on distributed memory parallel machines

Two protocols have been proposed [7,8] for the implementation of Gamma on network of communicating machines. The major problems to be solved are:

1. The detection of the tuples which may react.
2. The transformation of the multiset by application of reactions.
3. The detection of the termination.

The two proposed protocols differ in the way rewritings are controlled: in one case a central controller manages information transfers between processing elements; in the other case, information transfers are managed in a fully asynchonous way.

- **A synchronous Gamma machine**

  The protocol is based on a n-step (n being the cardinality of the argument multiset) algorithm. A controller C is connected to m processors $(P_0, \ldots, P_{m-1})$ and monitors information exchanges between these processors. The elements of the multiset are distributed over the local memories of P_i's. For the sake of simplicity, we assume that the number of processors equals the number of values in the multiset and that reactions and actions are binary. Each processor P_k possesses a variable i_k indicating the identity of its correspondent at a given step of the execution. Values i_k are initialised to $(n-k)$ *mod* n. After each computation step, the value of i_k is updated $(i_k := i_k + 1 \ mod \ n)$ thus providing a new configuration and producing a new set of pairs to which the reaction condition may be applied. Starting from any step, any sequence of n consecutive steps generates all possible configurations. This property is used to detect termination: the system can stop whenever $n-1$ configurations have been built without reaction.

The above synchronous implementation has been implemented on a Connexion Machine [25]. Other experiments have been conducted on the Maspar 1 SIMD machine: [43] describes an implementation of Gamma which results in a very good speed-up and a good exploitation of parallel resources. [42] shows how Higher-Order Gamma programs can be refined for an efficient execution on a parallel machine.

- **An asynchronous Gamma machine**

  The values of the multiset are spread over a chain of m processors. There is no central controller in the system and each processor knows only its two neighbours. Each processor manages three variables: A value v and two indexes: n_1, the number of exchanges undergone by v and n_2 the number of successive exchanges with processors possessing values v' with $n'_1 > n - 1$. It has been proven that if a processor possesses a value v with $n_2 \geq n - 1$, then this value will not be involved in any further reaction and the corresponding process can stop its activity. The termination detection algorithm is fully distributed over the chain of processors; however, the cost of this detection can be high compared with the cost of the computation itself [7,8].

  This solution has been implemented on an Intel iPSC2 machine [7,8] and on a Connexion Machine [25]. The results show a good exploitation of the processing power and speedup. However, a distributed operating system kernel should be developed in order to circumvent the hypotheses on the number of processors or on the number of values.

An implementation on shared memory machines

The Gamma model can be seen as a shared memory model: the multiset is the unique data structure from which elements are extracted and where elements resulting from the reaction are stored. It is clear that shared memory multiprocessors should be good candidates for parallel implementations of Gamma. A specific software architecture has been developed in [34] in order to provide an efficient Gamma implementation on a Sequent multiprocessor machine. Several techniques have been experimented in order to improve significantly the overall performances:

- The variables in the reaction condition are ordered in such a way that part of the condition can be tested as early as possible on the selected elements. If a partial test fails then the remaining elements can be ignored. This optimisation is similar to the technique described in [24].

- Operations defined in terms of arithmetic comparisons can be optimised by restricting the range of the bag elements which have to be considered. The equality operation is especially interesting to this respect. In order to implement this optimisation, the physical organisation of the multiset must reflect the arithmetic relation between elements. [34] proposes to use ordered trees and hashing tables to represent multisets. This idea is close to methods used for the implementation of relational data base systems.

- Redundant evaluations can be avoided by keeping track of already performed evaluations in a log which contains elements of the "same generation". Generation $n+1$ is processed only after generation n has been fully processed. This is achieved by keeping generation $n+1$ in an auxiliary multiset which is transferred in the original multiset only after generation n has been completely processed.

A kernel operating system has also been developed in order to cope with various traditional problems and in particular with the synchronisation required by Gamma (a multiset element cannot participate in more than one reaction at a time). Several experiments have been carried out on various examples. They all show the impact of the proposed optimisations on the performances which, even if difficult to compare with traditional imperative implementations, look very promising.

The same group has investigated the introduction of the Gamma paradigm in a functional language[39]. They introduce the multiset as a data structuring facility and multiset versions of operations $fold$ and map. Various implementations of these operations on a shared memory multiprocessor are investigated and evaluated.

A hardware implementation of Gamma based on tropes

The tropes defined in section 2.2. have been used as a basis for the design of a specialised architecture [56]. A hardware skeleton is associated with each trope and these skeletons are parameterised and combined according to the program to be implemented. A circuit can then be produced from a program description. The hardware platform is the PRL-DEC Perle 1 board which is built around a large array of bit-level configurable logic cells [11]. This array is surrounded by local RAM banks used as a cache, a programmable board and some additional logic to manage the host bus interface. Programming Perle 1 consists of describing an architecture using C++ and built-in primitives functions. The final objective of this work is to provide an environment which compiles Gamma programs (using tropes) into hardware (Perle 1 code). In a first step, manual translation from Gamma to Perle 1 was applied on some classical examples (like prime factorisation) and performances were compared with C implementations of the same problems on Sparc workstations. The Perle 1 implementation ranges between Sparc 10 and Sparc 2 implementations, showing that the proposed approach may lead to reasonable performances even for a high-level language like Gamma.

5 Conclusion

A number of languages and formalisms bearing similarities with the chemical reaction paradigm have been proposed in the literature. Let us briefly review the most significant ones:

- A programming notation called *associons* is introduced in [49]. Essentially an associon is a tuple of names defining a relation between entities. The state can be changed by the creation of new associons representing new relations derived from the existing ones. In contrast with Gamma, the model is deterministic and does not satisfy the locality properties (dues to the presence of $\forall$ properties).

- A Unity program [17] is basically a set of multiple-assignment statements. Program execution consists in selecting non deterministically (but following a fairness condition) some assignment statement, executing it and repeating forever. [17] defines a temporal logic for the language and the associated proof system is used for the systematic development of parallel programs. Some Unity programs look very much like Gamma programs (an example is the exchange sort program presented in the introduction). The main departures from Gamma is the use of the array as the basic data structure and the absence of locality property. On the other hand, Unity allows the programmer to distinguish between synchronous and asynchronous computations which makes it more suitable as an effective programming languages for parallel machines. In the same vein as Unity, the *action systems* presented in [6] are *do-od* programs consisting of a collection of guarded atomic actions, which are executed nondeterministicly so long as some guard remains true.

- Linda [33,18] contains a few simple commands operating on a tuple space. A producer can add a value to the tuple space; a consumer can read (destructively or not) a value from the tuple space. Linda is a very elegant communication model which can easily be incorporated into existing programming languages.

- LO [3,4,5] (for Linear Objects) was originally proposed as an integration of logic programming and object-oriented programming. It can be seen as an extension of Prolog with formulae having multiple heads. From an object-oriented point of view, such formulae are used to implement methods. A method can be selected if its head matches the goal corresponding to the object in its current state. The head of a formula can also be seen as the set of resources consumed by the application of the method (and the tail is the set of resources produced by the method). In [4], LO is used as a foundation for *interaction abstract machines*, extending the chemical reaction metaphor with a notion of broadcast communication: subsolutions (or "agents") can be created dynamically and reactions can have the extra effect of broadcasting a value to all the agents. A translation of a Gamma-like language into linear logic is presented in [16].

Taking a dual perspective, it is interesting to note that the physical modelling community has borrowed concepts from computer science leading to formalisms which bear similarities with higher-order Gamma. An example of this trend of activity is the "Turing gas" [30] where molecules float at random in a solution, reacting if they come into contact with each other. A language akin to lambda-calculus is used to express the symbolic computation involved in the reactions.

As a conclusion, we hope that this paper has shown that the chemical reaction paradigm is an active research topic. We can basically distinguish two main directions in the current works in this area:

1. The first one follows the seminal presentation of the cham [12] and explores the paradigm in a theoretical setting in order to provide better foundations for the definition of semantics for concurrent languages [1,38,41,47]. An uptodate survey of this trend of work can be found in [14].

2. The second one is more practically oriented and studies the usefulness of the paradigm for specific application domains [15,21,44,37,52] or tries to extend the formalism to make it more usable (in terms of efficiency and software engineering) [35,20,45,31].

A new illustration of the relevance of the paradigm for proving properties of distributed systems is provided in [13]. The context is the formal verification of cryptographic protocols. It is shown that these protocols can be expressed naturally in terms of chemical reactions on a multiset containing the states of the (trusted and untrusted) agents. As an example, the well-known Needham-Schroeder protocol is described as a collection of fourteen reactions.

Among the recent proposals for enhancing the basic version of Gamma, let us mention the following:

- The language of *schedules* [19,20] is a new approach for providing extra information about control in Gamma programs. The key idea is to separate the definition of a Gamma program in two parts: individual reactions, which correspond to a single application of a rewrite rule, and schedules, which specify the control part of the program. The language of schedules includes iteration, sequential and parallel composition, non-determinism. A nice property of schedules is that they disentangle the two orthogonal features of Gamma (the choice of multisets as the data structure and the "stirring mechanism" as the control structure) and they allow the user to make his own choice concerning the control component of the program.
- *Local linear logic* [45] is another interesting proposal for increasing the efficiency of Gamma without compromising its style. Local linear logic extends the principle of resource-consciousness of linear logic to a notion of locality-consciousness. Locality is expressed by associating indices to elements in a sequence and imposing that elements move towards their correct place in the sequence. In other words, the multiset is replaced by a self-ordering sequence.
- Also, the structured version of Gamma[31] described in section 3.4 seems to be a promising research direction with respect to implementation issues because it makes the structure of the data explicit and available to the compiler.

These proposals tackle the same problem in complementary ways: schedules introduce a way to master control when local linear logic and structured Gamma are two different means for structuring the multiset. The crucial issue is that these proposals were carefully designed in order to preserve the fundamental qualities of the formalism. So these efforts will hopefully converge towards a new generation of more practical languages based on the sounds principles of the chemical reaction paradigm.

Acknowledgments

We would like to express our thanks to all the people who have contributed to the development of Gamma during these years. They are too numerous to be named individually here but we hope that none of them has been forgotten in the bibliography. We are also grateful to the referees for their careful reading and valuable suggestions.

References

1. S. Abramsky, *Computational interpretations of linear logic*, Theoretical Computer Science, Vol. 111, pp. 3-57, 1993.
2. R. Allen and D. Garlan, *Formalising architectural connection*, Proceedings of the IEEE 16th International Conference on Software Engineering, pp. 71-80, 1994.
3. J.-M. Andreoli and R. Pareschi, *Linear Objects: logical processes with built-in inheritence*, New Generation Computing, Vol. 9, pp. 445-473, 1991.
4. J.-M. Andreoli, P. Ciancarini and R. Pareschi, *Interaction abstract machines*, in *Proc. of the workshop Research Directions in Concurrent Object Oriented Programming*, 1992.
5. J.-M. Andreoli, *Coordination in LO*, this volume.
6. R. Back, *Refinement calculus, part II: parallel and reactive programs*, in *Proc. of the workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, 1989, Springer Verlag, LNCS 430.
7. J.-P. Banâtre, A. Coutant and D. Le Métayer, *A parallel machine for multiset transformation and its programming style*, Future Generation Computer Systems, pp. 133-144, 1988.
8. J.-P. Banâtre, A. Coutant and D. Le Métayer, *Parallel machines for multiset transformation and their programming style*, Informationstechnik, Oldenburg Verlag, Vol. 2/88, pp. 99-109, 1988.
9. J.-P. Banâtre and D. Le Métayer, *The Gamma model and its discipline of programming*, Science of Computer Programming, Vol. 15, pp. 55-77, 1990.
10. J.-P. Banâtre and D. Le Métayer, *Programming by multiset transformation*, Communications of the ACM, Vol. 36-1, pp. 98-111, January 1993.
11. P. Bertin, D. Roncin and J. Vuillemin, *Programmable active memories: a performance assessment*, in *Proc. of the workshop on Parallel architectures and their efficient use*, 1992, Springer Verlag, LNCS , pp. 119-130.
12. G. Berry and G. Boudol, *The chemical abstract machine*, Theoretical Computer Science, Vol. 96, pp. 217-248, 1992.
13. D. Bolignano, *An Approach to the Formal Verification of Cryptographic Protocols*, *Proc. of the 3rd ACM conference on Computer and Communication Security*, 1996.
14. G. Boudol, *Some chemical abstract machines*, in *Proc. of the workshop on "a decade of concurrency"*, 1994, Springer Verlag, LNCS 803, pp. 92-123.

15. M. Bourgois, *Advantages of formal specifications: a case study of replication in Lotus Notes*, in Proc. of the conference on "Formal Models in Open Object-based Distributed Systems", March 1996.
16. P. Bruscoli and A. Guglielmi, *A linear logic view of Gamma style computations as proof searches*, this volume.
17. Chandy M. and Misra J., *Parallel program design: a foundation*, Addison-Wesley, 1988.
18. N. Carriero and D. Gelernter, *Linda in context*, Communications of the ACM, Vol. 32-4, pp. 444-458, April 1989.
19. M. Chaudron and E. de Jong, *Schedules for multiset transformer programs*, this volume.
20. M. Chaudron and E. de Jong, *Towards a compositional method for coordinating Gamma programs*, in *Proc. Coordination'96 Conference*, Cesena, 1996, Springer Verlag, LNCS, to appear.
21. P. Ciancarini, D. Fogli and M. Gaspari, *A logic language based on multiset rewriting*, this volume.
22. P. Ciancarini, R. Gorrieri and G. Zavattaro, *An alternative semantics for the calculus of Gamma programs*, this volume.
23. D. Cohen and J. Muylaert-Filho, *A calculus for higher-order multiset programming*, in *Proc. Coordination'96 Conference*, Cesena, 1996, Springer Verlag, LNCS, to appear.
24. C. Creveuil, *Techniques d'analyse et de mise en œuvre des programmes Gamma*, Thesis, University of Rennes, 1991.
25. C. Creveuil, *Implementation of Gamma on the Connection Machine*, in *Proc. of the workshop on Research Directions in High-Level Parallel Programming Languagea*, Mont-Saint Michel, 1991, Springer Verlag, LNCS 574, pp. 219-230.
26. C. Creveuil and G. Moguérou, *Développement systématique d'un algorithme de segmentation d'images à l'aide de Gamma*, Techniques et Sciences Informatiques, Vol. 10, No 2, pp. 125-137, 1991.
27. Dershowitz N. and Manna Z., *Proving termination with multiset ordering*, Communications of the ACM, Vol. 22-8, pp. 465-476, August 1979.
28. Dijkstra E. W., *The humble programmer*, Communications of the ACM, Vol. 15-10, pp. 859-866, October 1972.
29. L. Errington, C. Hankin and T. Jensen, *A congruence for Gamma programs*, in *Proc. of WSA conference*, 1993.
30. W. Fontana, *Algorithmic chemistry*, *Proc. of the workshop on Artificial Life*, Santa Fe (New Mexico), Addison-Wesley, 1991, pp. 159-209.
31. P. Fradet and D. Le Métayer, *Structured Gamma*, Irisa Research Report PI-989, March 1996.

32. S. Gay and C. Hankin, *A program logic for Gamma*, this volume.
33. Gelernter D., *Generative communication in Linda*, ACM Transactions on Programming Languages and Systems, Vol. 7,1, pp. 80-112, January 1985.
34. K. Gladitz and H. Kuchen, *Parallel implementation of the Gamma-operation on bags*, *Proc. of the PASCO conference*, Linz, Austria, 1994.
35. C. Hankin, D. Le Métayer and D. Sands, *A calculus of Gamma programs*, in *Proc. of the 5th workshop on Languages and Compilers for Parallel Computing*, Yale, 1992, Springer Verlag, LNCS 757.
36. C. Hankin, D. Le Métayer and D. Sands, *A parallel programming style and its algebra of programs*, in *Proc. of the PARLE conference*, Munich, 1993, Springer Verlag, LNCS 694, pp. 367-378.
37. P. Inverardi and A. Wolf, *Formal specification and analysis of software architectures using the chemical abstract machine model*, IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 373-386, April 1995.
38. A. Jeffrey, *A chemical abstract machine for graph reduction*, TR 3/92, University of Sussex, 1992.
39. H. Kuchen and K. Gladitz, *Parallel implementation of bags*, in *Proc. ACM Conf. on Functional Programming and Computer Architecture*, ACM, pp. 299-307, 1993.
40. D. Le Métayer, *Higher-order multiset programming*, in *Proc. of the DIMACS workshop on specifications of parallel algorithms*, American Mathematical Society, Dimacs series in Discrete Mathematics, Vol. 18, 1994.
41. L. Leth and B. Thomsen, *Some Facile chemistry*, TR 92/14, ECRC, 1992.
42. Lin Peng Huan, Kam Wing Ng and Yong Qiang Sun, *Implementing higher-order Gamma on MasPar: a case study*, Journal of Systems Engineering and Electronics, Vol. 16, No 4, 1995.
43. Lin Peng Huan, Kam Wing Ng and Yong Qiang Sun, *Implementing Gamma on MasPar MP-1*, Journal of Computer Science and Technology, to appear.
44. H. McEvoy, *Gamma, chromatic typing and vegetation*, this volume.
45. H. McEvoy and P.H. Hartel, *Local linear logic for locality consciousness in multiset transformation, Proc. Programming Languages: Implementations, Logics and Programs, PLILP'95*, Utrecht, 1995, Springer Verlag, LNCS 982, pp. 357-379.
46. R. Milner, *Communication and concurrency*, International Series in Computer Science, Prentice Hall, Englewood Cliffs, NJ, 1989.

47. R. Milner, *Functions as processes*, Mathematical Structures in Computer Science, Vol. 2, pp. 119-141, 1992.
48. L. Mussat, *Parallel programming with bags*, in *Proc. of the workshop on Research Directions in High-Level Parallel Programming Languagea*, Mont-Saint Michel, 1991, Springer Verlag, LNCS 574, pp. 203-218.
49. M. Rem, *Associons: a program notation with tuples instead of variables*, ACM Transactions on Programming Languages and Systems, Vol. 3,3, pp. 251-261, July 1981.
50. M. Reynolds, *Temporal semantics for Gamma*, this volume.
51. W.-P. de Roever, *Why formal methods are a must for real-time system specification*, in *Proc. Euromicro'92*, Panel discussion, June 1992, Athens.
52. H. Ruiz Barradas, *Une approche à la dérivation formelle de systèmes en Gamma*, Thesis, University of Rennes 1, July 1993.
53. D. Sands, *A compositional semantics of combining forms for Gamma programs*, in *Proc. of the Formal Methods in Programming and their Applications conference*, Novosibirsk, 1993, Springer Verlag, LNCS 735, pp. 43-56.
54. D. Sands, *Composed reduction systems*, this volume.
55. L. Van Aertryck and O. Ridoux, *Gammalog as goal-directed proofs*, internal report.
56. M. Vieillot, *Synthèse de programmes Gamma en logique reconfigurable*, Technique et Science Informatiques, Vol. 14, pp. 567-584, 1995.