Gamma and the chemical reaction model: fifteen years after*

Jean-Pierre Banâtre[1], Pascal Fradet[2], Daniel Le Métayer[3]

[1] Université de Rennes I, Campus de Beaulieu, 35042 Rennes, France
and INRIA, Domaine de Voluceau-Rocquencourt, 78153 Le Chesnay Cedex, France
jpbanatre@inria.fr

[2] INRIA / IRISA, Campus de Beaulieu, 35042 Rennes, France
fradet@irisa.fr

[3] Trusted Logic S.A. 5, rue du Bailliage, 78000 Versailles, France
Daniel.Le_Metayer@trusted-logic.fr

Abstract. 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 around Gamma considered as a programming or as a specification language. A special emphasis is placed on unexpected applications of the chemical reaction model, showing that this paradigm has been a source of inspiration in various research areas.

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. Sequentiality should no longer be seen as the prime programming paradigm but just as one of the possible forms of cooperation between individual entities.

* This paper is a revised version of *Gamma and the chemical reaction model: ten years after* [10]. It has been reorganized and includes additional sections on applications of the chemical reaction model. Sections presenting large examples, extensions of the formalism and implementations issues have been seriously shortened. The reader is referred to [10] and the original papers for further details on these topics.

The Gamma formalism was proposed fifteen years ago 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.

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

The side condition $x \leq y$ 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 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 refinement 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.

To conclude this introduction, let us quote E.W.Dijkstra thirty years ago [27]: *"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 monstrosities, 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.

We start by providing in section 2 the basic intuitions about the programming style entailed by Gamma as well as various programming examples and extensions.

Gamma was originally proposed in the context of a work on systematic program derivation [8]. Section 3 describes how Gamma can be used as an intermediate language in the derivation of efficient implementations from specifications.

Since the initial developments, Gamma has been a source of inspiration in unexpected research areas. Section 4 shows several applications of the chemical reaction model in various domains such as the semantics of process calculi, imperative programming and software architectures. Section 5 concludes by a sketch of related work.

2 Gamma as a programming language

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 favored 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. We proceed by presenting large applications written in Gamma and reviewing various linguistic extensions.

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 multiset 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.

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

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 [9]), 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:

$$\begin{array}{ll} fib(n) & = add(dec_1(n)) \\ dec_1 & = x \rightarrow x - 1,\ x - 2 \Leftarrow x > 1 \\ & \quad x \rightarrow 1 \Leftarrow x = 0 \\ add & = x,\ y \rightarrow x + y \Leftarrow True \end{array}$$

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.

$$\begin{array}{ll} maxss(M) & = max_g(max_l(M)) \\ max_l & = (i, x, s),\ (i', x', s') \rightarrow (i, x, s),\ (i', x', s + x') \\ & \quad \Leftarrow (i' = i + 1)\ and\ (s + x' > s') \\ max_g & = (i, x, s),\ (i', x', s') \rightarrow (i', x', s') \Leftarrow s' > s \end{array}$$

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. Five schemes (basic reactions), called *tropes* (for *t*ransmuter, *r*educer, *op*timiser, *e*xpander, *s*elector) are particularly useful. We present only three of them here:

- Transmuter.
$$\mathcal{T}(C, f) = x \rightarrow f(x) \Leftarrow C(x)$$
The transmuter applies the same operation to all the elements of the multiset until no element satisfies the condition.
- Reducer.
$$\mathcal{R}(C, f) = x, y \rightarrow f(x, y) \Leftarrow C(x, y)$$
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.
- **E**xpander.
$$\mathcal{E}(C, f_1, f_2) = x \rightarrow f_1(x), f_2(x) \Leftarrow C(x)$$
The expander is used to decompose the elements of a multiset into a collection of basic values.

The Fibonacci function can be expressed as the following combination of tropes:

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

The maximum segment sum program presented in the previous section can be defined in terms of tropes in a very similar way. Further details about tropes may be found in [35].

2.3 Larger applications

The interested reader can find in [9] 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 application. Gamma has been used in a project aiming at experimenting high-level programming languages for prototyping image processing applications [24]. The application was the recognition of the tridimensional topography of the vascular cerebral network from two radiographies. 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.

Another example of application of Gamma to image processing is reported in [47]. 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.

Reactive programming. In [58], an operating system kernel is defined in Gamma and proven correct in a framework inspired by the Unity logic [14]. An important result of this work is the definition of a temporal logic for 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.

2.4 Implementations

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. 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.

This section sketches several attempts to provide parallel implementations for Gamma programs.

Distributed memory implementations Two protocols have been proposed [6, 7] for the implementation of Gamma on network of communicating machines. They differ in the way rewritings are controlled:

- *Centralized control.* The elements of the multiset are distributed over the local memories of the processors. A central controller is connected to all the processors and monitors information exchanges. This protocol has been implemented on a Connexion Machine [23]. Other experiments have been conducted on the Maspar 1 SIMD machine: [46] describes an implementation of Gamma which results in a very good speed-up and a good exploitation of parallel resources. [45] shows how Higher-Order Gamma programs can be refined for an efficient execution on a parallel machine.
- *Distributed control.* Information transfers are managed in a fully asynchronous way. 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 neighbors. 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 [6, 7]. This solution has been implemented on an Intel iPSC2 machine [6, 7] and on a Connexion Machine [23]. The results show a good exploitation of the processing power and speedup.

Shared memory implementations. The Gamma model can also 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. Shared memory multiprocessors are good candidates for parallel implementations of Gamma. A specific software architecture has been developed in [33] 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. A kernel operating system has also been developed in order to cope with various traditional problems and in particular with the synchronization required by Gamma (a multiset element cannot participate in more than one reaction at a time).

Hardware implementation. The tropes defined in section 2.2 have been used as a basis for the design of a specialized architecture [62]. A hardware skeleton is associated with each trope and these skeletons are parameterized and combined according to the program to be implemented. A circuit can then be produced from a program description. The hardware platform was the PRL-DEC Perle 1 board which is built around a large array of bit-level configurable logic cells [11].

2.5 Linguistic extensions

The Gamma programs that we have presented so far are made from a single block of reaction rules. In this section, we review several linguistic extensions for structuring programs or multisets.

Composition operators for Gamma. 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. Several proposals which

have been made to extend Gamma with facilities for building complex programs from simple ones.

[34] 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.

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

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 disjoint 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 *synchronized 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 new vision of parallel composition and its combination with the sequential composition creates interesting semantical problems. [34] defines a set of program refinement and equivalence laws for parallel and sequential composition, by considering the input-output behavior 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.

Several compositional semantics of the language have also been proposed [59, 20]. Not all the laws established in the operational semantics remain valid in these semantics; this is because they distinguish programs with identical input/output behavior but which behave differently in different contexts. It is shown however that most interesting properties still hold, the great advantage of a compositional semantics being that laws can be used in a modular way to prove properties of large programs. Other semantics of Gamma have been proposed, including [28] which defines a congruence based on transition assertions and [55] which describes the behavior of Gamma programs using Lamport's Temporal Logic of Actions.

Composition operators have been studied in a more general framework called *reduction systems* [60] 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.

Higher-order Gamma. 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 [41] 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 4.1).

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:

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

E_1 is a multiset containing the active configuration $[P_1, M = M_0]$ initialy. 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 $.E_2$.

[41] 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 generalization of the chemical model to higher-order is presented in [21].

Structured Gamma. 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 have to 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 jeopardizing the basic qualities of the language. 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 ruin the fundamental locality principle (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. As an example, the list [5; 2; 7] can be represented by a structured multiset whose set of addresses is $\{a_1, a_2, a_3\}$ and associated values (written $\overline{a_i}$) are $\overline{a_1} = 5$, $\overline{a_2} = 2$, $\overline{a_3} = 7$. Let **next** be a binary relation and **end** a unary relation; the addresses satisfy

$$\textbf{next } a_1\ a_2,\ \textbf{next } a_2\ a_3,\ \textbf{end } a_3$$

A new notion of type is introduced in order to characterize precisely the structure of the multiset. A type is defined in terms of a graph grammar (or rewrite rules).

A structured multiset belongs to a type T if its underlying set of addresses satisfies the invariant expressed by the grammar defining T. As an example, the list type can be defined by the following context-free graph grammar:

$$\begin{array}{l} List = L\ x \\ L\ x\ = \textbf{next}\ x\ y,\ L\ y \\ L\ x\ = \textbf{end}\ x \end{array}$$

Any multiset which can be produced by this grammar belongs the the $List$ type. Reading the grammar rules from right to left gives the underlying rewrite system. So alternatively, any multiset which can be reduced using this rewrite rules 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 **end** x is a one element list. A circular list can be defined as follows:

$$\begin{array}{ll} Circular = L\ x\ x & \\ L\ x\ y & = L\ x\ z,\ L\ z\ y \\ L\ x\ y & = \textbf{next}\ x\ y \end{array}$$

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 addresses.

Here are some examples of programs operating on lists:

$$\begin{array}{lll} Sort\ : List = \textbf{next}\ a\ b & \rightarrow \textbf{next}\ a\ b,\ a\ :=\ \overline{b}, b\ :=\ \overline{a} & \Leftarrow \overline{a} < \overline{b} \\ Mult : List = \textbf{next}\ a\ b,\ \textbf{next}\ b\ c & \rightarrow \textbf{next}\ a\ c, a\ :=\ \overline{a} * \overline{b} & \\ Iota\ : List = \textbf{end}\ a & \rightarrow \textbf{next}\ a\ b,\ \textbf{end}\ b, b\ :=\ \overline{a} - 1 & \Leftarrow \overline{a} > 1 \end{array}$$

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 **next** relation, but $Mult$ shrinks the list by removing the intermediate element b.

The natural question following the introduction of a new type system concerns the design of an associated type checking algorithm. In the context of Structured Gamma, type checking must ensure that a program maintains the underlying structure defined by a type. It amounts to the proof of an invariant property. There exists a sound checking algorithm based on the construction of an abstract reduction graph. For a reaction $C\ \rightarrow\ A$ and type T, the algorithm

computes the possible contexts X such that $X + C$ reduces to $\{T\}$ (i.e. belongs to $\{T\}$). It is then sufficient to check that $X + A$ (i.e. the multiset after the reaction) reduces to $\{T\}$.

Structured Gamma allows the programmer to define his own types and have his programs checked according to the type definitions. 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. Applications of this approach to imperative programming and the analysis of software architectures are described in sections 4.2 and 4.3. 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.

3 Gamma as a bridge between specifications and implementations

In the previous section, we presented Gamma as a programming language and tried to convey the programming style entailed by the chemical reaction model through a series of examples. Gamma can be also seen as a very high-level language bridging the gap between specification languages and low-level (implementation oriented) languages.

3.1 From specifications to Gamma programs

We first present the techniques that can be used to prove properties of Gamma programs. Then, we suggest how they can be used to derive programs from specifications in a systematic way.

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 the introduction. 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 [26] 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 mimics the behavior 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 systematically. 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 [8]. The basic strategy consists in splitting the specification into a conjunction 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 initialization 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 *primes* in section 2.1. The input multiset is $\{2, \ldots, n\}$ and a possible specification of the result M is the following:

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

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:

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

The interested reader can find a more complete treatment of several examples in [8]. A slightly different approach is taken in [52] 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 shortest paths, the maximum segment sum and the longest up-sequences 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.

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).

3.2 From Gamma programs to efficient implementations

As mentioned earlier, the philosophy of Gamma is to introduce a clear separation between correctness issues and efficiency issues in program design. In particular, Gamma can be seen as a specification language which does not introduce unnecessary sequentiality. As a consequence, designing a reasonably efficient implementation of the language is not straightforward. This section outlines several

optimizations allowing Gamma programs to be refined into efficient, sequential programs.

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 optimize the execution and produce acceptable performances. In his thesis, C. Creveuil [22] studied several optimizations which are summarized here.

One important source of inefficiency comes from useless (redundant or deemed to fail) checks of the reaction condition. Three optimizations 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 neighborhood 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. [22] 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 [22] is that well-known efficient versions of sequential algorithms (shortest path for instance) can be "rediscovered" and justified as the result of several optimizations of a naïve implementation of Gamma. It is very often the case that the most drastic optimizations rely on structural properties of the values belonging to the multiset (neighborhood relationship, ordering in the choice of values ...).

Such properties are difficult to find automatically and the optimizations described above can be seen as further refinement steps rather than compilation techniques. 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. For example, the language of *schedules* [17, 18] provides extra information about control in Gamma programs, and *local linear logic* [48] as well as Structured Gamma [31] structure the multiset.

4 Gamma as of source of inspiration

The chemical reaction model has served as the basis of a number of works in various, often unexpected, research directions. In particular, ideas borrowed from Gamma have been applied to process calculi, imperative programming and software architectures. We describe these three developments in turn and conclude with a sketch of a few other proposals. Most of these works are quite significant and open new research directions but it is important to note that none of them jeopardizes 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".

4.1 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"* [13]. 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 sub-solution $\{m_1, \ldots, m_n\}$ 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 [50] 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:

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

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

  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 $\mid$ 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 [13]. The Cham has inspired a number of other contributions. Let us mention some of them:

- [1] uses a *linear Cham* to describe the operational semantics of proof expressions for the classical linear logic.
- [51] defines an operational semantics of the π-calculus in a Cham style.
- [39] describes a graph reduction in terms of a Cham.
- [43] 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 modeling concurrency which is an essentially associative and commutative notion. As stated in [12]: *"In the SOS style of semantics, labeled 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 closer to the execution process of sequential languages, or the evaluation mechanism of functional languages [13].

4.2 Shape types

Type systems currently available for imperative languages are too weak to detect a significant class of programming errors. The main reason is that they fail to capture properties about the sharing which is inherent in many data structures used in efficient imperative programs. As an illustration, it is impossible to express the property that a list is doubly-linked or circular in existing type systems. The work around Structured Gamma (section 2.5) showed that many data structures could be described as graph grammars and manipulated by reactions. Furthermore, a static algorithm can be used to check that a reaction preserve the structure specified by the grammar. These ideas and techniques have been adapted in order to extend the type system of C and make pointer manipulation safer [30].

Shape-C is an extension of C which integrates the notion of types as graph grammars (called here *shapes*) and reactions. The notion of graph grammars is powerful enough to describe most complex data structures (see [30] for a description of skip lists, red-black trees, left-child-right-sibling trees in terms of graph grammars).

The design of Shape-C was guided by the following criteria:

- the extensions should be blended with other C features and be natural enough for C programmers,
- the result of the translation of Shape-C into pure C should be efficient,
- the checking algorithm of section 2.5 should be applicable to ensure shape invariance.

We present Shape-C through an example: the Josephus program. This program, borrowed from [56], first builds a circular list of n integers; then it proceeds through the list, counting through $m-1$ items and deleting the next one, until only one is left (which points to itself). Figure 1 displays the program in Shape-C. The Josephus program first declares a shape `cir` denoting a circular list of integers with a pointer `pt`. Besides cosmetic differences, the definition of shapes is similar to the context free grammars presented in Section 2.5. The variables are interpreted as addresses. They possess a value whose type must be declared (here `int`). Values can be tested or updated but cannot refer to addresses. They do not have any impact on shape types.

Intuitively, unary relations (here `pt`) correspond to roots whereas binary relations (here `next`) represent pointer fields. Shapes can be translated into C structures with a value field and as many fields (of pointer type) as the shape has binary relations.

```
/* Integer circular list                                                    */
shape int cir { pt x, L x x;
                L x y = L x z, L z y;
                L x y = next x y;     };
main()
{int i, n, m;
/*initialization to a one element circular list                            */
 cir s =  [| => pt x; next x x; $x=1; |];
 scanf("%d%d", &n, &m);
/* Building the circular list 1->2->...->n->1                              */
 for (i = n; i > 1; i--)
     s:[| pt x; next x y; => pt x; next x z; next z y; $z=i; |];
/* Printing and deleting the m th element until only one is left           */
 while (s:[| pt x; next x y; x != y; => |])
 {
   for (i = 1; i < m-1; ++i)
       s:[| pt x; next x y; => pt y; next x y; |];
   s:[| pt x; next x y; next y z; => pt z; next x z; printf("%d ",$y); |];
 }
/* Printing the last element                                                */
 s:[| pt x => pt x;  printf("%d\n",$x); |];
}
```

Fig. 1. Josephus Program

Shape-C uses only a subset of graph grammars that corresponds to the rooted pointer structures manipulated in imperative languages. This subset is defined by the following properties:

(S1) *Relations are either unary or binary.*
(S2) *Each unary relation is satisfied by exactly one address in the shape.*
(S3) *Binary relations are functions.*
(S4) *The whole shape can be traversed starting from its roots.*

These conditions allow shapes to be implemented by simple C structures (with a value and pointer fields). They can be enforced by analyzing the definition of grammars.

The reaction, written `[| C => A |]`, is the main operation on shapes. Two specialized versions of reactions are also provided: initializers, with only an action, written `[| => A |]` and tests, with only a condition, written `[| C => |]`.

The Josephus program declares a local variable `s` of shape `cir` and initializes it to a one element circular list.

```
cir s =  [| => pt x; next x x; $x = 1; |];
```

The value of address `x` is written `$x` and is initialized to 1. In general, actions may include arbitrary C-expressions involving values. The for-loop builds a n element circular list using the reaction

```
s:[| pt x; next x y; => pt x; next x z; next z y; $z=i; |];
```

The condition selects the address `x` pointed to by `pt` and its successor. The action inserts a new address `z` and initializes it to `i`. The translation in pure C is local and applied to each shape operation of the program. Shape-C enforces a few simple restrictions on reactions so that the translation is both direct and efficient.

Shape checking amounts to verify that initializations and reactions preserve the shape of objects. The checking algorithm is directly based on the algorithm outlined in section 2.5. Note that values and expressions on values are not relevant for shape checking purposes. Using this algorithm, it is easy to ensure that the list `s` is cyclic throughout the Josephus program.

Due to their precise characterization of data structures, shape types are a very useful facility for the construction of safe programs. Most efficient versions of algorithms are based on complex data structures which must be maintained throughout the execution of the program [16, 56]. The manipulation of these structures is an error-prone activity. Shape types permits to describe invariants of their representation in a natural way and have them automatically verified.

4.3 Software architectures

Another related area of application which has attracted a great amount of interest during the last decade 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 engineering discipline of software is a formal basis for describing and analyzing these designs"*. Typical examples of software architectures are the "client-server organization", "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 organization of computational entities and the interactions between them [2]. The chemical reaction model has been used for specifying software architectures [38] and architecture styles [42].

Software architecture specification. The application considered in [38] is a multi-phase compiler and two architectures are defined using the "chemical abstract machine" [12, 13]. 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:

$$\begin{array}{c} i(d_1) \ \diamond \ o(d_2) \ \diamond \ m_1, \ o(d_1) \ \diamond \ m_2 \ \diamond \ i(d_3) \\ \rightarrow \\ o(d_2) \ \diamond \ m_1 \ \diamond \ i(d_1), \ m_2 \ \diamond \ i(d_3) \ \diamond \ o(d_1) \end{array}$$

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:

$$\begin{array}{c} i(tok) \ \diamond \ o(phr) \ \diamond \ parser, \ o(tok) \ \diamond \ lexer \ \diamond \ i(char) \\ \rightarrow \\ o(phr) \ \diamond \ parser \ \diamond \ i(tok), \ lexer \ \diamond \ i(char) \ \diamond \ \ o(tok) \end{array}$$

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, [38] defines a correspondence between multisets generated by two versions of the multi-phase compiler and establishes a form of bisimulation between the two architectures. They also prove normalization properties of the programs.

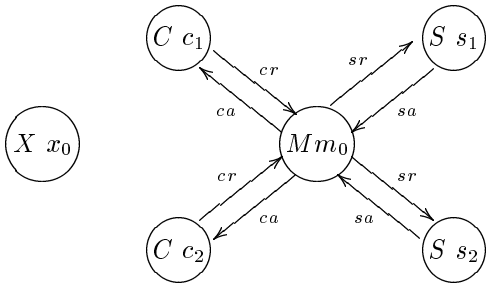
Software architecture styles. The approach described in [42] focuses on the interconnection between individual components of the software architecture. The main goal is to describe architecture styles (i.e. classes of architectures) and to check that the dynamic evolution of an architecture preserves the constraints imposed by the style. Techniques developed for Structured Gamma (graph grammars and the associated checking algorithm) can be applied to this problem.

Structured Gamma allows connections to become "first class" objects and to prove invariance properties on the structure of the network. For example, a client-server architecture style can be defined as the graph grammar

$$\begin{array}{ll} ClientServer & = CS \ m \\ CS \ m & = \mathbf{cr} \ c \ m, \mathbf{ca} \ m \ c, \ \mathbf{C} \ c, \ CS \ m \\ CS \ m & = \mathbf{sr} \ m \ s, \ \mathbf{sa} \ s \ m, \ \mathbf{S} \ s, \ CS \ m \\ CS \ m & = \mathbf{M} \ m, \ \mathbf{X} \ x \end{array}$$

The unary relations **C**, **S**, **M** and **X** correspond respectively to client, server, manager and external entities. The external entity stands for the external world;

it records requests for new clients wanting to be registered in the system. The binary relations **cr** and **ca** correspond to client request links and client answer links respectively (**sr** and **sa** are the dual links for servers). For example, the architecture



involves two clients c_1 and c_2, two servers s_1 and s_2, a manager m_0 and the external entity x_0. It belongs to the client-server class (grammar) *ClientServer*.

It is often the case that the architecture of an application should be able to evolve dynamically. For instance, a client-server organization must allow for the introduction of new clients or their departure, a pipeline may grow or shrink, facilities for dealing with mobile computing may be required. In this framework, the evolution of the architecture is defined by a *coordinator*. The task of the coordinator is expressed by reaction rules. As an illustration, the following coordinator applies to a client-server architecture:

$$\mathbf{X}\ x,\ \mathbf{M}\ m \rightarrow \mathbf{X}\ x',\ \mathbf{M}\ m,\ \mathbf{cr}\ c\ m, \mathbf{ca}\ m\ c,\ \mathbf{C}\ c$$

$$\mathbf{cr}\ c\ m, \mathbf{ca}\ m\ c,\ \mathbf{C}\ c\ \rightarrow \emptyset$$

The two rules[1] describe respectively the introduction of a new client in the architecture and its departure. The main benefit of this approach is that the algorithm of section 2.5 can be used to ensure that a coordinator does not break the constraints of the architecture style. This makes it possible to reconcile a dynamic view of the architecture with the possibility of static checking. For example, had we forgotten, say **cr** $c\ m$ in the right-hand side of the first rule, then the coordinator would have been able to transform a client-server architecture into an architecture which would not belong any longer to the class defined by *ClientServer*.

The interested reader can find in [37] the treatment of an industrial case study proposed by the Signaal company [25] using a multiple views extension of the formalism presented above. The goal of the work was the specification of a railway network system. The static verification of coordination rules with respect to grammars was deemed the most attractive feature of the formalism [37].

These applications provide evidence that (Structured) Gamma is an intuitive and formally based formalism to describe and analyze software architectures.

[1] In fact, these rules are completed with side conditions on the states of the entities otherwise, the coordinator could add or remove entities without any consideration of the current state of the system.

4.4 Other works

Influences of the chemical reaction model can be found in other domains such as visual languages [36], protocols for shared virtual memories [49] or logic programming [19, 61]. We just sketch here works around coherence protocols and logic programming where Gamma played a key rôle.

Formalization of coherence protocols. Coherence protocols for shared virtual memories have been formalized as Gamma programs in [49]. The multiset, whose elements are the protocol entities or events, represents a global view of the system. The protocol itself is described as a Gamma program (i.e. a collection of reaction rules). Besides, a fragment of first-order logic is used to specify properties that the protocol is expected to satisfy. This formalization made it possible to design an algorithm checking that properties are indeed invariants of the protocol. [49] presents a Gamma formalization of the Li and Hudak protocol [44] as well as the automatic verification of a collection of invariants.

This approach has been applied to software architectures in [53]. The dynamic evolution of the architecture is described as a Gamma program (as in section 4.3). Instead of checking membership to a given style (as in section 4.3), the algorithm of [49] is reused to check that the evolutions of the architecture respect some logical properties specified in a separate language.

Gamma and logic programming. Several proposals have been made for integrating Gamma and logic programming languages. A first approach, followed in [19], uses multisets of terms and describes conditions and actions as predicates. This model is implemented as an extension of Gödel, a strongly typed logic programming language with a rich module system. 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 extension, called Gammalög, includes the sequential and parallel composition operators introduced in [34]. Another approach is followed in [61] 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 [3] which can itself be seen as an elaboration on the basic chemical reaction model. 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.

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 [54]. 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 [14] 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. [14] 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 [5] are *do-od* programs consisting of a collection of guarded atomic actions, which are executed nondeterministically so long as some guard remains true.
- Linda [32, 15] 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 (for Linear Objects) was originally proposed as an integration of logic programming and object-oriented programming [3]. 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: sub-solutions (or "agents") can be created dynamically and reactions can have the extra effect of broadcasting a value to all the agents.

Taking a dual perspective, it is interesting to note that the physical modeling 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" [29] 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 model is a particularly simple and fruitful paradigm. No doubt that new surprising developments are yet to come.

Acknowledgments We would like to express our thanks to all the people who have contributed to the development of Gamma during these years.

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. 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.
6. 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.
7. 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.
8. 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.
9. J.-P. Banâtre and D. Le Métayer, *Programming by multiset transformation*, Communications of the ACM, Vol. 36-1, pp. 98-111, January 1993.
10. J.-P. Banâtre and D. Le Métayer, *Gamma and the chemical reaction model: ten years after*, in Coordination Programming: Mechanisms, Models and Semantics, Imperial College Press, 1996.
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. G. Boudol, *Some chemical abstract machines*, in *Proc. of the workshop on A decade of concurrency*, 1994, Springer Verlag, LNCS 803, pp. 92-123.
14. Chandy M. and Misra J., *Parallel program design: a foundation*, Addison-Wesley, 1988.
15. N. Carriero and D. Gelernter, *Linda in context*, Communications of the ACM, Vol. 32-4, pp. 444-458, April 1989.
16. T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to algorithms*, MIT Press, 1990.
17. M. Chaudron and E. de Jong, *Schedules for multiset transformer programs*, in Coordination Programming: Mechanisms, Models and Semantics, Imperial College Press, 1996.
18. M. Chaudron and E. de Jong, *Towards a compositional method for coordinating Gamma programs*, in *Proc. Coordination'96 Conference*, Lecture Notes in Computer Science, Vol. 1061, pp. 107-123, 1996.

19. P. Ciancarini, D. Fogli and M. Gaspari, *A logic language based on multiset rewriting*, in Coordination Programming: Mechanisms, Models and Semantics, Imperial College Press, 1996.
20. P. Ciancarini, R. Gorrieri and G. Zavattaro, *An alternate semantics for the calculus of Gamma programs*, in Coordination Programming: Mechanisms, Models and Semantics, Imperial College Press, 1996.
21. D. Cohen and J. Muylaert-Filho, *Introducing a calculus for higher-order multiset programming*, in *Proc. Coordination'96 Conference*, Lecture Notes in Computer Science, Vol. 1061, pp. 124-141, 1996.
22. C. Creveuil, *Techniques d'analyse et de mise en œuvre des programmes Gamma*, Thesis, University of Rennes, 1991.
23. C. Creveuil, *Implementation of Gamma on the Connection Machine*, in *Proc. of the workshop on Research Directions in High-Level Parallel Programming Languages*, Mont-Saint Michel, 1991, Springer Verlag, LNCS 574, pp. 219-230.
24. 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.
25. E. de Jong, *An industrial case study: a railway control system*, Proc. Second int. Conf. on Coordination Models, Languages and Applications, Springer Verlag, LNCS 1282, 1997.
26. Dershowitz N. and Manna Z., *Proving termination with multiset ordering*, Communications of the ACM, Vol. 22-8, pp. 465-476, August 1979.
27. Dijkstra E. W., *The humble programmer*, Communications of the ACM, Vol. 15-10, pp. 859-866, October 1972.
28. L. Errington, C. Hankin and T. Jensen, *A congruence for Gamma programs*, in *Proc. of WSA conference*, 1993.
29. W. Fontana, *Algorithmic chemistry*, *Proc. of the workshop on Artificial Life*, Santa Fe (New Mexico), Addison-Wesley, 1991, pp. 159-209.
30. P. Fradet and D. Le Métayer, *Shape types*, in Proc. of Principles of Programming Languages, POPL'97, ACM Press, pp. 27-39, 1997.
31. P. Fradet and D. Le Métayer, *Structured Gamma*, Science of Computer Programming, 31, pp. 263-289, 1998.
32. Gelernter D., *Generative communication in Linda*, ACM Transactions on Programming Languages and Systems, Vol. 7,1, pp. 80-112, January 1985.
33. K. Gladitz and H. Kuchen, *Parallel implementation of the Gamma-operation on bags*, *Proc. of the PASCO conference*, Linz, Austria, 1994.
34. 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.
35. C. Hankin, D. Le Métayer and D. Sands, *A parallel programming style and its algebra of programs*, in *Proc. of the PARLE conference*, LNCS 694, pp. 367-378, 1993.
36. B. Hoffmann. *Shapely Hierarchical Graph Transformation*, Symposium on Visual Languages and Formal Methods (VL FM'01) in the IEEE Symposia on Human-Centric Computing Languages and Environments (HCC'01), IEEE Press, 2001.
37. A. A. Holzbacher, M. Périn and M. Südholt, *Modeling railway control systems using graph grammars: a case study*, Proc. Second int. Conf. on Coordination Models, Languages and Applications, Springer Verlag, LNCS 1282, 1997.
38. 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.

39. A. Jeffrey, *A chemical abstract machine for graph reduction*, TR 3/92, University of Sussex, 1992.
40. H. Kuchen and K. Gladitz, *Parallel implementation of bags*, in *Proc. ACM Conf. on Functional Programming and Computer Architecture*, ACM, pp. 299-307, 1993.
41. 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.
42. D. Le Métayer, *Describing software architecture styles using graph grammars*, IEEE Transactions on Software Engineering (TSE), Vol. 24(7), pp. 521-533, 1998.
43. L. Leth and B. Thomsen, *Some Facile chemistry*, TR 92/14, ECRC, 1992.
44. K. Li, P. Hudak, *Memory Coherence in Shared Virtual Memory Systems*, in Proc. of ACM Symposium on Principles of Distributed Computing, pp. 229-239, 1986.
45. 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(4), 1995.
46. Lin Peng Huan, Kam Wing Ng and Yong Qiang Sun, *Implementing Gamma on MasPar MP-1*, Journal of Computer Science and Technology.
47. H. McEvoy, *Gamma, chromatic typing and vegetation*, in Coordination Programming: Mechanisms, Models and Semantics, Imperial College Press, 1996.
48. H. McEvoy and P.H. Hartel, *Local linear logic for locality consciousness in multiset transformation*, *Proc. Programming Languages: Implementations, Logics and Programs, PLILP'95*, LNCS 982, pp. 357-379, 1995.
49. D. Mentré, D. Le Métayer, T. Priol, *Formalization and Verification of Coherence Protocols with the Gamma Framework*, in Proc. of the 5th Int. Symp. on Software Engineering for Parallel and Distributed Systems (PDSE-2000), ACM, 2000.
50. R. Milner, *Communication and concurrency*, International Series in Computer Science, Prentice Hall, Englewood Cliffs, NJ, 1989.
51. R. Milner, *Functions as processes*, Mathematical Structures in Computer Science, Vol. 2, pp. 119-141, 1992.
52. L. Mussat, *Parallel programming with bags*, in *Proc. of the workshop on Research Directions in High-Level Parallel Programming Languages*, Mont-Saint Michel, 1991, Springer Verlag, LNCS 574, pp. 203-218.
53. M. Périn, *Spécifications graphiques multi-vues : formalisation et vérification de cohérence*, PhD thesis, Université de Rennes 1, 2000.
54. M. Rem, *Associons: a program notation with tuples instead of variables*, ACM Trans. on Programming Languages and Systems, Vol. 3,3, pp. 251-261, 1981.
55. M. Reynolds, *Temporal semantics for Gamma*, in Coordination Programming: Mechanisms, Models and Semantics, Imperial College Press, 1996.
56. R. Sedgewick, *Algorithms in C*, Addison-Wesley publishing company, 1990.
57. W.-P. de Roever, *Why formal methods are a must for real-time system specification*, in *Proc. Euromicro'92*, Panel discussion, June 1992, Athens.
58. H. Ruiz Barradas, *Une approche à la dérivation formelle de systèmes en Gamma*, Thesis, University of Rennes 1, July 1993.
59. 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.
60. D. Sands, *Composed reduction systems*, in Coordination Programming: Mechanisms, Models and Semantics, Imperial College Press, 1996.
61. L. Van Aertryck and O. Ridoux, *Gammalog as goal-directed proofs*, internal report.
62. M. Vieillot, *Synthèse de programmes Gamma en logique reconfigurable*, Technique et Science Informatiques, Vol. 14, pp. 567-584, 1995.