A Distributed Implementation of *Structured Gamma*

Gabriel A. L. Paillard
Universidade Tiradentes
paillard@cos.ufrj.br

Felipe M. G. França
Programa de Engenharia de Sistemas e Computação, Universidade Federal do Rio de Janeiro
felipe@ieee.org

Juarez A. Muylaert Filho
Instituto Politécnico Universidade Estadual do Rio de Janeiro
jamf@iprj.uerj.br

Abstract

This paper presents a distributed implementation of the Structured Gamma *programming language, a language based on the* Gamma *multiset rewriting paradigm.* Structured Gamma *offers, in addition to the advantages introduced by* Gamma, *implicit concurrent behavior and a type system where not only types themselves are defined but also the automatic verification of user defined types at compilation time. Problems and mechanisms involved in a* MPI-*based implementation of* Structured Gamma *using a type checking engine based on* MGU – Most General Unifier *are investigated.*

1. Introduction

Parallel programming may not be an easy task, especially if the programming language to be used doesn't have resources that allow separating the problem to be solved, from its implementation. The *Gamma* language was created as a way to abstract only about the problem to be solved, leaving aside precepts belonging, for example, to the imperative paradigm, that make the task of creating parallel programs harder. Jean-Pierre Banâtre and Daniel Le Métayer created gamma in 1986, as a formalism to specify programs, based in the multiset parallel rewriting, making the proofs of correctness and derivations of programs easier [1].

The main feature of a Gamma program is the free interaction between elements of the multiset which makes the execution model non-deterministic because there is no restriction on how the data elements are to be manipulated by the program rules (in Gamma one says that they can "react" freely), leaving the programs to be naturally and implicitly executed in parallel.

Aiming at making Gamma a more practical programming language in which structured data types such as lists could be dealt with, the authors of Gamma proposed a extension of it called *Structured Gamma* [2], whilst keeping the original multiset rewriting paradigm. The main goal of this work is to present the problems tackled and the theory involved in a parallel and distributed implementation of Structured Gamma.

The reasons that took the making of this implementation are not focussed only in the search of performance improvement, but also in the search of a robust and more expressive programming style in the development of parallel programs.

The Gamma multiset rewriting programming paradigm, including clear examples, is explained in the next section. Section III presents implicit concepts and potentialities of Structured Gamma as a programming language. The problems and mechanisms involved in the implementation of Structured Gamma are exposed in section IV. Section V presents our conclusions.

2. Gamma

The widespread sequential paradigm can not be, of course, considered as the only one in program development; rather, it better be seen as one of possible ways of cooperation between individual computational entities. The Gamma formalism was proposed fourteen years ago to describe computing as a global evolution of atomic values that interact freely [3]. A nice way to introduce Gamma is by the *chemical reaction* metaphor. The only data model in Gamma is the *multiset* (which is "like" a set where members can have an arity greater then one). It can be seen as a chemical solution, in which data items are the reacting parts of the solution.

A simple program is a multiset together with an basic reaction, which can be thought of a programming procedure or function, to be executed on the multiset elements

v1, v2, ..., vn $\rightarrow$Action $\Leftarrow$ Reaction Condition

where v1, v2, ..., vn are to be bound to some elements (that are removed at the time of binding) from the multiset, the *Reaction Condition* being a boolean

expression, possibly involving v1, v2, ..., vn, which when true allows the *Action* to be executed which is a list of arithmetic expressions, possibly involving v1, v2, ..., vn, which specifies those values that are to be inserted into the multiset.

The program execution consist in substituting the multiset elements that satisfy the reaction condition by the list of values specified in the action until a so-called *stable state* is reached, i.e., when any reaction can be performed anymore because either there aren't enough elements to react or the elements don't react.

Then, it's desirable to have an abstract algorithm of the following way:

"*While there is at least two elements in the multiset, select two elements from it, compare them and remove the minor from it.*"

Below, we present an example of a Gamma program, using the operator Γ, that computes the greatest element in a non-empty multiset of integers numbers.

$$\text{Maxconj}(S) = \Gamma((R,A))(S) \text{ where}$$
$$R(x,y) = x \leq y$$
$$A(x,y) = y$$

The function R specifies the property to be satisfied by the two selected elements (earlier defined as "reaction condition"). These elements are substituted in the multiset by the result of the application of the function A (earlier defined as "action").

Now, let's give a formal presentation for Gamma. The data model in Gamma (**G**eneral **A**bstract **M**odel for **M**ultiset m**A**nipulation) is the multiset, similar to a set, except that it may contain multiple occurrences of the same element. The advantage of using multiset is the possibility to describe compounded data with any form of hierarchy between the components. This is not the case of recursively defined data structures, like lists, that imposes order in the examination of their elements. The control structure associated to multisets is the operator Γ. As we saw in the example above, Gamma reflects the absence of hierarchy in the data models and implies in a probabilistic execution model. Its formal definition can be seen by the following way:

$$\Gamma((R_1,A_1),..,(R_m,A_m))(M) =$$
$$\quad \textit{if } \forall\, i \in [1,m], \forall\, x_1,..,x_n \in M,\ \neg R_i(x_i,..,x_n)$$
$$\quad \textit{then } M$$
$$\quad \textit{else let } x_1,..,x_n \in M, \textit{let } i \in [1,m] \textit{ such that } R_i(x_1,..,x_n) \textit{ in}$$
$$\qquad \Gamma((R_1,A_1),..,(R_m,A_m))((M-\{x_1,..,x_n\}) + A_i(x_1,..,x_n))$$

The effect of a reaction (R_i, A_i) over a multiset M is to replace M in a subse of elements, $(X_1, ..., X_n)$, so that $R_i(X_1, ..., X_n)$ is true, by the elements from $A_i(X_i, ..., X_n)$. If no element from M satisfies any reaction $((\forall\, i \in [1,m], \forall\, x_1,..,x_n \in M, \neg R_i(x_1,..,x_n))$, then the result is the same M. Otherwise, the result is obtained realizing a reaction $((M-\{x_1,..,x_n\}) + A_i(x_1,..,x_n))$ and repeating the same process until no reaction can occur. This definition implies that if one or more reaction condition can be verified for some subsets at the same time, the choice made between them is not deterministic. The importance of the locality property can be underestimated. If the reaction condition is satisfied by some disjoint subsets, the reactions can be realized simultaneously and independently. This property is the basic reason for the potential parallelism of Gamma programs [4].

3. Structured Gamma

Some deficiencies in Gamma were verified. The main ones are [2]:

- the original Gamma definition doesn't have any operation to combine programs;
- Gamma doesn't make easy the task to structure data or to specify a control strategy;
- for the combinatorial explosion imposed by its semantics, it's hard to reach a good level of efficiency in any Gamma implementation.

The impossibility to create data structure in Gamma and the difficult to impose a particular control strategy are according with Gamma original motivation, which was to create programs with the least number of restrictions. However, the problem is that sometimes it was necessary to use difficult strategies to implement its algorithm. We may use as an example, the case where we had to represent pairs in a multiset (index, value). This limitation also introduces an inefficiency factor in the implementation, because the data structure (and control) cannot seen by the compiler. Such information could be used to improve the implementation.

A structured multiset consists in addresses satisfying relations, what may be seen as a kind of neighborhood between the reaction molecules, using the chemical reaction analogy. Therefore, the locality aspect, one of the main Gamma characteristics, continues being preserved, because we have a topological sight of the multiset, avoiding the manipulation of it as one entire piece, what comes to favor the parallelism.

The list [5, 2, 7] can be represented by a structured multiset, which set of addresses is $\{a_1, a_2, a_3\}$ and their values are, $Val(a_1) = 5$, $Val(a_2) = 2$, $Val(a_3) = 7$, we may express the values using the following notation: $\overline{a_3}$, that has the same meaning of $Val(a_3)$. Consider Succ a binary relation and End an unary relation; the addresses satisfy:

$$\text{Succ } a_1\ a_2,\ \text{Succ } a_2\ a_3,\ \text{End } a_3$$

Therefore, we have a list representation, in which the element a_1 precedes a_2, and a_3 is the tail of the list. A structured Gamma program continues to be, then, a series

of reaction applications realized locally, i.e., Gamma paradigm keeps being the same.

A Gamma structured program is defined based in pairs compounded by a condition and an action, which can:

- test or modify the relations over the addresses;
- test or modify the associated values with the addresses.

A structured multiset M may be seen as M = Rel + Val, where:

- Rel is the multiset of relations, represented by triples of the kind (Val, r, a) (r ∈ R, a ∈ A);
- Val is a set of values represented by triples of the kind (Val, a, v) (a ∈ A, v ∈ V).

For example, the structured multiset showed in the beginning can be represented by the following way:
{(Succ, a_1, a_2), (Succ, a_2, a_3), (End, a_3), (Val, a_1, 5), (Val, a_2, 2), (Val, a_3, 7)}

If a structured multiset has an address x, this won't have more than one value, i.e., x will appear only once in Val. In the other hand, it may exist some occurrences of the same tupla in Rel. We also didn't focussed the fact that:

$$A(Rel) \subseteq A(Val) \text{ nor } A(Val) \subseteq A(Rel).$$

Then, we may conclude that each allocated address may not have a value, or can have a value, but they don't figure in any relation. In that case, a structured Gamma program can't access them and they can be garbage collected.

The structured Gamma program semantics ($P = C_1 \Rightarrow A_1, ..., C_m \Rightarrow A_m$) applied to a multiset M is defined as the set of normal forms from the following rewriting system:

$$M \to_P gc(M) \text{ if } \forall \{x_1,.., x_n\} \subseteq A(M)\ \forall\, i \in [1,..,m] \neg T(C_i)(x_i,.., x_n)$$

$$M \to_P M - C(C_i)(x_1,.., x_n) + A(A_i)(x_1,.., x_n, y_1,.., y_k) (\text{with } y_1,.., y_k \notin A(M))$$
$$\text{if } \exists \{x_1,.., x_n\} \subseteq A(M) \text{ and } \exists i \in [1,..,m] \text{ such that } T(C_i)(x_1,.., x_n)$$

If no address tuple satisfies any condition, then the normal form is found. The result is the structure. Addresses that do not occur in Rel are remove from Val. i.e. :

$gc(Rel+Val) = Rel + \{(val, a, v) \mid (val, a, v) \in Val \wedge a \in A(Rel)\}$

A changer (program) substitutes an instance of its left side (condition C) by an instance of its right side (action A) [5]. To synthesize the formalism given above, what happens is that a tuple of addresses (x_i, ..., x_n) and a pair (C_i, A_i) such that $T(C_i)$ (x_i, ..., x_n) are not deterministically chosen. The multiset is transformed removing $C(C_i)$ (x_i, ..., x_n) and allocating new addresses (y_i, ..., y_k), and adding in multiset $A(A_i)(x_1, ..., x_n, y_1, ..., y_n)$.

An example of a structured multiset goes right below. It can be schematized by a graph, as showed in Fig. 1.
M = {p a1, pred a1 a1, next a1 a2, pred a2 a1, next a2 a3, pred a3 a2, next a3 a3}

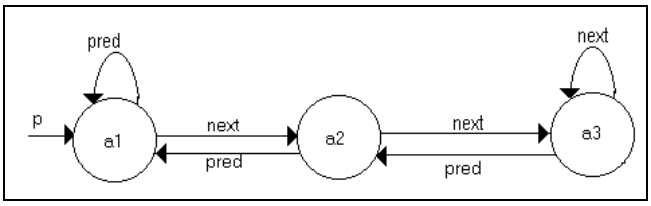


Fig.1 A double linked list.

Structured multiset are a syntactical facility used to make the organization of the data explicitly [2]. Now, we'll introduce the notion of a shape, what validates the structure defined in a multiset. We define a shape by the rewriting rules over the multisets. A multiset will belong, or will be according to determined shapes, if its relations set may be produced by the rewriting system, that defines the shape.

In a structured Gamma program, we'll have a user defined shape, what will use a free context grammar, giving the possibility of the definition of various data structure in a very simple way. Also will be supplied as an input data for this program a structured multiset, what was defined before.

The shape syntax is defined by the following grammar:
<Type Declaration> = TypeName = [<Prod>], [<Non Terminal> = <Prod>]*
<Non Terminal> = NTName x_1,..,x_n
<Prod> = *r* x_1,..,x_n | Non Terminal | <Prod>

Where r ∈ R is a relation, and x_i is a variable that has a determined address. A shape definition is similar to a free context grammar. In the sequence, we'll show an example of binary tree shape that can be defined by the following way:

Bintree = B_x
B_x = Left x y, Right x z, B_y, B_z
B_x = Leaf x x

The grammar above represented, allows that a rule produces consequently one element (node), which can have two children or be a leaf, what characterizes a strict binary tree.

In imperative languages, like C, when we declare two types as a tree and a linked list, we'll only be able to distinguish these two kinds by their respective functions. In structured Gamma, the type shape is incorporated in the declaration of the own type and a routine that inserts one element in a double linked list, if it's not correct, will do some kind of error (in compile-time). Because of that, structured Gamma is strongly typed.

A grammar consists of four components (NT, T, PR, O), where NT and T are the non-terminal and terminal nodes, respectively. PR is the set of production rules and O is the origin of the derivation. The production rules are pairs l = r, in which l is a non-terminal node and r is a

collection of elements. Therefore, the shape representing a double linked list is:

H_{DL}= <{Doubly, L}, {Next, Pred, p}, R_{Doubly}, Doubly>

After this initial phase, it was developed a type verification algorithm. A type verification algorithm associates to each defined type in the structured Gamma program, a rewriting system over the relations of the multiset. For the previously defined type (double linked lists), the rewriting system is:

p x, pred xx, Lx $\rightarrow$ R_{Doubly} Doubly

next x y, pred y x, Ly,X $\rightarrow$ R_{Doubly} Lx, X $y \notin X$

next x x, X $\rightarrow$ R_{Doubly} Lx, X

We used the Greek letter σ to represent variable substitution, which is an injective function that will be used in the rewriting system. A shape defined by a grammar is the set:

{M|M $\rightarrow^{*}$ PR {O} and M terminal}

X + (σ) $\rightarrow$ PR X + (σl)

l=r $\in$ PR e (Var(σl)) $\cap$ Var(X) = $\varnothing$

A multiset belongs to a shape if, using the rewriting system (by the use of the production rules), it's possible to get to the initial symbol of the grammar. Another alternative consists in producing a set containing the multiset derived from the origin "O", but it's a more expensive way.

It's easy to verify that the multiset given below, which is an example of a double linked list, belongs to the shape H_{DL}. This can be observed in a simple way:

M = {p a_1, pred a_1 a_1, next a_1 a_2, pred a_2 a_1, next a_2 a_3, pred a_3 a_2, next a_3 a_3}

Applying the rewriting system to the multiset, we have:

next a_3 a_3, X $\rightarrow$ L_{a3}, X

next a_2 a_3, pred a_3 a_2, L_{a3}, X $\rightarrow$ L_{a2}, X $a_3 \notin$ X

next a_1 a_2, pred a_2 a_1, L_{a2}, X $\rightarrow$ L_{a1}, X $a_2 \notin$ X

p a_1, pred a_1 a_1, L_{a1} $\rightarrow$ Doubly

Then, we have the confirmation that the given multiset satisfies the type defined by the shape H_{DL}. X is used to represent the reduction context, in which variables that disappeared from the right side of the production rules can't show in the rest of the reduction, anymore.

4. Implementation of Structured Gamma

The first phase of Structured Gamma implementation consisted, obviously, in the construction of the lexical analyzer, in which are recognized the language words that aren't in huge quantity. In the second phase, it was realized the syntactical analysis that verifies, by the construction of a grammatical tree, the syntactical structure of the program. These two phases of the compiler were developed using Yac, a tool that comes with Lex [6]. The generated grammar is non-ambiguous. About the generated data structure (grammatical tree) by the syntactical analysis, it was made the type verification and the code generation. To make a better idea of this phase, let's have an example, using the syntax that was implemented:

shape:

<{Doubly, L}, (next, pred, p), Doubly>

tipo:

Doubly = p (x), pred (x x), L (x)

L x = next (x y), pred (y x), L (y)

L x = next (x x)

Programa:

P_1 {p (a_1), pred (a_1 a_1), next (a_1 a_2), pred (a_2 a_1), next (a_2 a_3), pred (a_3 a_2), next (a_3 a_3) valores: (a_1 := 5), (a_2 := 3), (a_3 := 6) } where

P_1 = p (a), next (a b), pred (b a) (b == a) $\Rightarrow$ p (a), next (a e), pred (e a), next (e b), pred (b e) (e := b)

The program above has as shape a double linked list, and its action consists in inserting one element after the first element in the list. The tree mounted in this phase has some dozens of pointers, being impossible to show here an illustration of it. The first token to be recognized by the grammar is "shape:". After that, comes between < > the non-terminal ones, terminal ones and the grammar origin symbol, respectively. After recognizing the token "tipo:", it'll examine and will put, in the adequate position of the syntactical tree, the free context grammar rules. And, finally, after token "Programa:", we'll have in first place the name of the programs, in the case of our example P_1, but also, we could have, for example, P_1 | P_2; P_3, what means that P_1 and P_2 will be executed in parallel. Token ";" means that P_3 will be run after P_1 and P_2 finish.

Each reaction will be set to a process, in the case of the operators referenced above, each one will also be transformed in a respective process. If we have P_1 | P_2, the "|" will be a process that will have the function to control execution of processes equivalent to reactions P_1 and P_2.

We also can, in the program declaration area, specifically in action, to make use of attributions, to alter the values of variables that already exist or new ones dynamically created. For example, a_1 := a_2, in which the value stored in address a_1 has its contents changed for the value contained in the address a_2. In condition, we also can have value comparisons. For example, a_1 = = a_2, i.e., if the contents of the given addresses are identical, it will return an affirmative answer. We have, beyond the one showed here, the other known relational operators.

4.1. Type Verification

Together with the creation of new system types, it must exist an algorithm for type verification. At Structured Gamma context, the algorithm must ensure that the executed program keep the data structure defined

in the type, i.e., that it doesn't degenerate. This is an invariant property of the system [2].

The first part of the type verification must be applied to the given multiset, because it must guarantee that it's according the grammar. The difference, when related to the algorithm that we'll see later, is that it's necessary to reach the grammar origin symbol. Using the rewriting system for a double linked list, we have:

$Check_{C,A}(PR,O) = Verify_A\ (Build_C(PR,O))$ where:

- $Build_C(PR,O)$ returns a tree with root C, condition C represents the initial condition given in the program, and the nodes $C_i \xrightarrow{X_i} C_{i+1}$, such that:

$\exists l = r \in PR, \exists \sigma \in UMG(C_i,(l,r))$ e

$X_i = (\sigma\ r) - C_i$

$C_{i+1} = (c_i - (\sigma\ r)) + (\sigma\ l)$

*a*nd C_{i+1} is not isomorphic when related to its predecessors C_j in the tree returned by the program;

- $Verify_A$ (Tree) returns true, if and only if,

$\forall C_i \xrightarrow{X_i} C_2 ... \xrightarrow{X_{k-1}} C_k$ complete path in the generated tree, $C_1 = C$ and C_k is a leaf,

$A + X_1 + ... + X_{k-1} \xrightarrow{*} PR\ C_k;$

- UMG(C,(l,r)) is the set of all substitutions σ from left variables and from right of production rules (PR), such that:

$C \cap (\sigma\ r) \neq \varnothing$ e $(Var(\sigma\ r) - Var(\sigma\ l)) \cap Var(G(\sigma\ r)) \neq \varnothing$

The implemented algorithm tries to build paths in the tree, from the given condition. If it doesn't reach a non-terminal, then it erases the traced path and afterwards, tries to build another path. Consequently, we may have more than one valid non-terminal, after the tree construction, but all of them must be tested by the action, and result in the same non-terminals. For the action, another tree is generated. After comparing both trees, and a result being found, validation or not of the program, according to the specified type in the grammar. It must be observed that, in the action, we can use as a node value, one element ∅, because the fundamental thing is to get to the same non-terminal reached in the condition.

Let's consider a practical example. Applying the rewriting paradigm to the double linked list one has:

p x, pred x x, Lx → R_{Doubly} Doubly

next x y, pred y x, Ly, X → R_{Doubly} Lx, X, y ∉ X

next x x, X → R_{Doubly} Lx, X

Observe that, in the second rule, y ∉ X. This has to be ensured in order to avoid variable sharing. The definition of the type double linked list is:

Doubly = p x, pred x x, Lx

Lx = next x y, pred y x, Ly

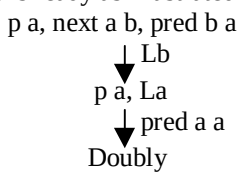
Lx = next x x

P1 = p a, next a b, pred b a ⇒

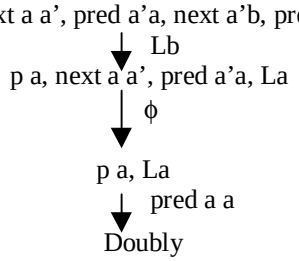
p a, next a a', pred a' a, next a' b, pred b a'

Note that P1 do an action that inserts an element at the beginning of a double linked list. A condition given by the program is that: p a, next a b, pred b a. Hence, using the type checking algorithm, one has that the reduction X1 = (next a b, pred b a, Lb) – (p a, next a b, pred b a), what will result in Lb. The first step is to obtain σr, what was done via applying the rewriting system to the double linked list. One can easily see that the condition matches with the rule next x y, pred y x, Ly, X → R_{Doubly} Lx, X, y ∉ X. After the first step of the type checking algorithm, one has two contexts: the first one is Lb, the reduction context, and the second one is the remaining of the condition, p a in this case. The restriction that a variable does not exist at the left side of the production rule applies to the last context.

The next step of the type checking algorithm is to find C1 that is equal to ((p a, next a b, pred b a) – (next a b, pred b a, Lb)) + (La), what will result in p a,La. Proceeding with the construction of the reduction tree, one has X_2 = (p a, pred a a, La) – (p a, La),that will result in pred a a. The new derived condition is ((p a, La) – (p a, pred a a, La)) + (Doubly), that will result in Doubly. The reduction tree is ready as illustrated below:



To conclude the type checking algorithm one has only to check the action defined by the program against the reduction contexts, what should lead to the same non-terminal and proving that the shape is kept invariant:



Observe that, one can use the f element as the edge value since is essential to arrive at the same non-terminal reached by the condition.

The role of MGU — Most General Unifier —[7] is described next. As the multiset variables are different from the ones found at the program and, by their turn, those are different from the ones defined by the grammar, there must exist a way to associate the mall. Remember that we are dealing with an injective function.

We have applied MGU to do that. A *disagreement* set is created in order to realize the matching of variables. Consider as example the following rule:

Lx = pred x y

If there is a pred a1 a2 in the multiset then the disagreement set would be {x|a1, y|a2}. MGU would use this disagreement set to perform the substitutions due. By matching a rule, MGU does the possible substitutions of the non-terminal under investigation. One has also to check if the differing elements, between non-terminals and terminals of the rule, are not present at the remaining context (X). This has been implemented by constructing a linked list where, at each element of the list, a disagreement set is found and kept for each iteration of the program.

In the code generation cells are created. The cell 0 has the function to send multiset copies to the other ones. Each program, i.e., each transform (C $\Rightarrow$ A) become a cell, which is responsible for the multiset change, and, after that, returns the same to the cell 0.

We may have processes connected to operators as ";" that specifies a sequence to be followed by the processes, i.e., the processes after the ";" will have to wait the finish of the previous ones, so that they begin their executions.

We used a message change interface in a distributed system. Each cell receives a program copy and executes its part. The cell 0 will end the program, when no more reactions may occur.

More exactly, cells are created, at execution time, each one being associated to a specific process. However, there will always be two cells, i.e., cell 0 and cell 1. Cell 0 is responsible for the management of the multiset, that is, to accept the requests for accessing the multiset and receiving it back after the execution of reactions have been performed by other cells. Cell1 is responsible for the execution of the other cells, including terminating them..

5. Conclusions

We have implemented a new parallel language, following a new paradigm (multiset rewriting). In the beginning of the implementation work [8] we have made a careful investigation towards checking if there was any implementation of Structured Gamma and none was found. In the implementation process, the Most General Unifier [7], was implemented and several extensions were purposed in this work. One can also find the proofs for the termination of the type verification algorithm in [2], where it was also implemented successfully. We have firstly shown the theoretical aspects of Structured Gamma and the potential advantages of its implementation, such as easy creation of data structures and the guarantee of having the type defined in a free context grammar, kept during all the program execution trough type verification.

At last, we implemented the code generator. The compiler output is a C program. As we have already stated, each reaction is associated to a specific process. The message interchange interface used was MPI (Message Parsing Interface), version LAM from Ohio 6.1 [9], which has proved to be very practical since the only need when moving from a distributed platform to another is to change parameters particular to the target system.

We were really content with the work, because our goal, which was to supply a implementation of a new parallel language, was reached. We intend to go on by purposing some extensions, and implementing them, giving, by this way, continuity to our work.

Acknowledgements

Finally, we would like to thank Othon Batista for helping in the definition of the first version of this paper and Claudio Prata for his help at the first stages in the implementation of Structured Gamma. This work was partially supported by CNPq and CAPES, Brazilian research Agencies.

References

[1] BANÂTRE, J.-P., LE MÉTAYER, D, A New Computational Model and its Discipline Programming, Rapport de recherche INRIA, nº 566, septembre 1986.

[2] FRADET, P., LE MÉTAYER, D., Structured Gamma, Irisa Research Report PI-989, March 1996.

[3] BANÂTRE, J.-P., LE MÉTAYER, D., Gamma and the Chemical Reaction Model: Ten Years After, Coordination Programming Mechanisms, Models and Semantics, Imperial College Press, 1996.

[4] BANÂTRE, J.-P., LE MÉTAYER, D. Programming by Multiset Transformation, Communication of the ACM, Vol. 36-1, pp. 98-111, January 1993.

[5] FRADET, P., LE MÉTAYER, D., Shape Types, Proc. ACM SIGPLAN Symposium on Principles of Programming Languages, pp. 27-39, 1997.

[6] LEVINE, John R., MASON Tony, BROWN, D., Lex & Yacc, United States of America, Editor Dale Dougherty, 1992.

[7] CASANOVA, Marco A., GIORNO, Fernando C., FURTADO, Antonio L., Programação em Lógica e a Linguagem Prolog, Capítulo 4, Rio de Janeiro, Brasil, Editora Edgard Blücher Ltda, 1987.

[8] PAILLARD, G.A.L., A Parallel and Distributed Implementation of Structured Gamma, MSc Thesis, PESC/COPPE, Universidade Federal do Rio de Janeiro, September 1999.

[9] GDB/RDB, MPI Primer Developing with LAM, 1996, The Ohio State University.