Lecture Notes in Computer Science 1061
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Advisory Board: W. Brauer D. Gries J. Stoer

Paolo Ciancarini Chris Hankin (Eds.)

Coordination Languages and Models

First International Conference
COORDINATION '96
Cesena, Italy, April 15-17, 1996
Proceedings

Springer

Volume Editors

Paolo Ciancarini
University of Bologna, Department of Computer Science
Pza. di Porta S. Donato, 5, I-40127 Bologna, Italy

Chris Hankin
Imperial College, Department of Computing
180, Queen's Gate, London SW7 2BZ, United Kongdom

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Coordination languages and models : first international conference, coordination '96, Cesena, Italy, April 15 - 17, 1996 ; proceedings / Paolo Ciancarini ; Chris Hankin (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Budapest ; Hong Kong ; London ; Milan ; Paris ; Santa Clara ; Singapore ; Tokyo : Springer, 1996
(Lecture notes in computer science ; Vol. 1061)
ISBN 3-540-61052-9
NE: Ciancarini, Paolo [Hrsg.]; GT

CR Subject Classification (1991): D.1.3, C.2.4, F1.2, D.2-4

ISBN 3-540-61052-9 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author
SPIN 10512685 06/3142 - 5 4 3 2 1 0 Printed on acid-free paper

Foreword

A new class of models, formalisms, and mechanisms for describing concurrent and distributed computations has emerged over the last few years. A characteristic feature of members of this class is that they are based on (generative) communication via a shared data space. They are called *coordination* languages and models.

This volume contains the proceedings of the First International Conference on Coordination Models and Languages (COORDINATION'96), held in Cesena (Italy) 15-17 April 1996.

In response to the call for papers, 78 papers were submitted to COORDINATION'96. All submitted papers were reviewed by at least 3 reviewers. The programme committee met at Imperial College (London) on 11 December 1995 and selected 21 regular papers. A further 10 papers were selected as short papers, to be presented at a poster session; these are included in this volume after the regular papers.

The programme committee invited Jean-Pierre Banâtre, Ugo Montanari, and Peter Wegner to give invited talks; these are included in this volume before the regular papers.

We thank all members of the programme committee and their sub-referees; they are listed on the following pages. We would also like to thank Roberto Gorrieri, the local arrangements chairperson, and Juarez Muylaert Filho and David Cohen for their assistance in processing the referees' reports. The following organisations provided sponsorship for the conference: Fondazione Cassa di Risparmio di Cesena, Italian National Research Council (C.N.R.), Comune di Cesena, Provincia di Forlì-Cesena, Olidata, Sun Microsystems, Silicon Graphics, Ascom TCS Safnat S.p.A., Link s.r.l., Libreria Minerva, and Cremonini Fabio s.r.l.. Finally, we would not have had the inspiration for arranging this conference had it not been for the EU-funded project COORDINATION; the project has provided partial financial support for a number of the European programme committee members.

April 1996 Paolo Ciancarini and Chris Hankin

Programme Committee

Gul Agha, University of Illinois, US
Jean-Marc Andreoli, Xerox Research Center Meylan, FR
Marc Bourgois, ECRC Munich, DE
Luca Cardelli, Digital SRC Palo Alto, US
Paolo Ciancarini, University of Bologna, IT
Laurent Dami, Université de Genève, CH
David Garlan, Carnegie Mellon University, US
David Gelernter, Yale University, US
Chris Hankin, Imperial College, UK (Chair)
Jose Meseguer, SRI International, US
Daniel Le Métayer, INRIA/IRISA Rennes, FR
Oscar Nierstarsz, Universitaet Bern (IAM), CH
António Porto, Uninova, Lisboa, PT
David Sands, DIKU, Copenhagen, DK
Akinori Yonezawa, University of Tokyo, JP

Local Arrangements

Roberto Gorrieri, Bologna, IT

List of Referrees

Birger Andersen
Andrea Asperti
Mark Astley
Uwe Borghoff
Paolo Bottoni
Patricia Bournai
Luis Caires
Pierre-Yves Chevalier
Juan Carlos Cruz
Jose Cunha
Henrique Joao Domingos
Steven Eker
Nabiel Elshiewy
Alessandro Fabbri
Daniela Fogli
Pascal Fradet
Markus Fromherz
Thom Fruehwirth
D. Galmiche
Mauro Gaspari
Natalie Glance
A. Gordon
Vineet Gupta
A.A. Holzbacher
S. Hunt
Valerie Issarny
R. Jagannathan
Nadeem Jamali
Jean-Marc Jezequel
J.N. Kok
Tsung-Min Kuo
Cosimo Laneve
Niels Elgaard Larsen
Patrick Lincoln
Markus Lumpe
Willem Mallon
Narciso Marti-Oliet
Cecilia Mascolo
T.D. Meijler
Antonio Messina
Luis Monteiro
Gilles Muller
Brian Nielsen
Jacques Noye
D. Pagani
Jens Palsberg
Remo Pareschi
Anna Patterson
Sophie Pinchinat
Noel Plouzeau
Shangping Ren
Philippe Rerole
M Reynolds
Olivier Ridoux
M. Riveill
Marco Roccetti
Eva Rose
Davide Rossi
Jean-Guy Schneider
Kees Schuerman
Scott Smith
Daniel Sturman
Jean-Pierre Talpin
Gunnar Teege
David N. Turner
Vasco Vasconcelos
J.Y Vion-Dury
Jan Vitek
James Waldby
Gianluigi Zavattaro
Lenore Zuck

Contents

Invited Papers

Regular Papers

Short Papers

Parallel Multiset Processing: From Explicit Coordination to Chemical Reaction

Jean-Pierre Banâtre

Irisa / Universiti de Rennes 1 / Inria
Campus de Beaulieu
35042 Rennes cedex - France

Abstract:

The author has been involved for more than fifteen years in the design, study and implementation of coordination program structures. These structures were designed with a clear conviction that data structuring and program structuring were two closely related issues. Very early, it was recognized that set data structuring was a key concept for the design of programs with a high potential for parallelism. This paper offers a personal perspective of this research activity which culminated with the Gamma formalism.

1 Introduction

Parallelism is a powerful structuring concept that could facilitate program construction. As stated in [3], two kinds of parallelism have to be distinguished : physical parallelism and logical parallelism. Physical parallelism is related to the organisation of the computation on a set of processors. By logical parallelism, we mean the possibility of describing a program as a set of cooperating processes. In this paper, we are only concerned with "logical parallelism".

The confusion between these two kinds of parallelism comes from the heritage of several decades of imperative culture and the impact of the Von Neumann model of computation. The sequencing operator together with assignment are the basic means of abstracting the mode of operation of the Von Neumann machine. They have been further developed with the introduction of the loop control structure and of the array data structure. Let us take the simple example of computing the maximum element of a set of values. In an imperative setting, the set will be represented by an array a $[1 : n]$ and a possible solution will be :

$$\begin{aligned} \text{max\_set} : & \; m := a[1]; \\ & \; i := 1; \\ & \; *[i < n \rightarrow i := i + 1; m := max(m, a[i])] \end{aligned}$$

while the condition $i < n$ holds, index i is incremented and a new local maximum is computed.

Two decisions have led to this highly sequential program : the choice of representing a set by an array, thus introducing an ordering between elements, and the use of a loop program structure for computing i and m.

The maximum element can in fact be computed by performing the comparisons of the elements in any order : every "confrontation" between two elements cancels the smaller one, and the unique remaining element will be the maximum. As will be seen later, this sentence describes an algorithm which can be readily written in Gamma [3].

The above example raises another crucial point. There is a very strong correlation between control structures and data structures. One knows that a variable, say v, represents a sequence whose successive elements verify a recurrence relationship of the form : $v_i = \varphi(v_{i-1})$, φ representing the body of the loop which computes v. It is also well-known that recursive data structures (like lists) are naturally exploited by using recursive control structures. In this paper, we will examine how appropriate control structures may be designed in order to carry out computations on general data structures such as multisets. Our aim will be the design of high-level programs which reflect the logical properties of the problem to solve, without any artificial sequentiality.

Section 2 describes coordinations structures which have been developed in order to process dynamic sets. Section 3 discusses the evolution from these structures to the Gamma formalism which is sketched in section 4.

2 Coordination structures for dynamic multiset processsing

This section gives an overview of work which was carried out fifteen years ago in a compiler design project [1,2]. In this context, it was recognized that proper control structures over proper data structures clearly enabled the user to better express the solutions of complex programming problems.

2.1 The notion of event

An event (called "future" in other contexts) can be seen as a single assignment variable [1]. The value of such variables may be requested in a computation before being actually produced. In this case, the computation demanding the value is simply suspended ; it is resumed when the value is known. Of course, the introduction of events comes with the possibility of dynamically creating processes. Here is a sample of program involving events and processes :

begin
(1) **event int** x;
...
(2) **activate** $(\cdots x \cdots)$; # process P_1#
...
(3) $x \leftarrow 3$
(4) **activate** $(\cdots x \cdots)$; # process P_2#
...
end

An event whose value is of type **integer** is declared in (1). In (2), the process P_1 is created. It will happen to be suspended, if x has not been assigned a value when it is accessed. The value 3 is assigned to x in (3) and finally a new process (say P_2) is created in (4). This process will not be interrupted because of x.

The concept of event has been largely used to simply solve the problem of forward referencing while constructing an Algol 68 compiler [1].

Another particular hard problem to deal with in the construction of compilers and operating systems is the management of dynamic identifier tables, i.e., tables whose cardinality cannot be known statically. Awkward sequential solutions are usually used (based on a priori fixed cardinality, which may reveal to be badly chosen or using dynamic data structures such as lists). Examining the very nature of these tables, it was clear that they would be better represented as sets rather than as arrays. Proceeding this way, they would also be naturally exploited in a parallel way.

Furthermore, the contents of these tables is built incrementally and their components can be seen as events whose values are determined as the computation progresses.

2.2 Dynamic sets of events

A set (say, s) of events of type **event m** is declared as **vse m** s ; *vse* stands for "varisized set of events". The generation of a new component belonging to s is expressed by $s \leftarrow v$, which means that one of the components of s gets its value, v. The cardinality of dynamic sets is itself an event. The value of this event is determined when some condition is realized during the computation. The primitive **close** s is used to signify that the *vse* s is completed.

Now we know how to declare and build *vse's*, but we haven't said yet how they can be exploited.

The control structure which was proposed in [2] acts as a process generator. As soon as a new element is introduced in a vse, s, new processes are generated which deal with s. Syntactically, this program structure can be described as :

for all elts of s **do** < process body > **od**

This peculiar loop acts as a process generator : when a new element s_j is added to s, an instance of < process body > dealing with s_j is spawn. In < process body >, s_j is referred to as **this** s ; this is simply a generic naming facility. Of course, several process generators may be associated to the same *vse* and elements s_j will, in general, be exploited by a "bunch" of processes.

In order to complete the above control structure, a coordination facility was added ; it allows the creation of an "epilogue" process, when **all** processes created by a given process generator have been completed. So the final format of our control structure is :

for all elts of s do < process body > **od**
at end < epilogue > **end**

More informations about *vse's* and associated control structures may be found in [2]. Next, we present an example illustrating the elegance of these data and program structures.

2.3 A general identifier binder

We consider the problem of associating an identifier to its declarations in a block structured language following the usual Algol/Pascal rules. For simplicity sake, we use the following grammar to describe the language :

< block > ::= **begin $ beg $** < body > **end $ ed $**
< body > ::= < sentence > | < sentence >;< body >
< sentence > ::= **var** < identifier > **$ var** ident $ |
< identifier > **$ occur** ident $|
< block >
< identifier > ::= # usual notation for identifiers #

A block is a sequence of declarations and instructions. An instruction is either the occurrence of a variable or a nested block.

Symbols between $ are transmitted by the parser to the semantic analyser which performs identifier binding. Symbols produced by the parser are accessed sequentially by successive requests to the process *rsd* (for read syntactic data).

The solution we propose can be written as follows :

```
procpidentification = ((identifier → void) enclosing_block_search) void :
  begin
   vse identifier id_table ;
   procp search = (identifier x )void :
     begin
      activate
       (bool success := false ;
       for all elt of id_table
         do
          [x = this id_table → success := true
            □
          x ≠ this id_table → null
          ]
         od
       at end
         [ not success → enclosing_block_search (x)
           □
          success → print (x, "successfull binding")
         ]
       end
       )
     end search ;
  *[rsd ? symbol →
     [symbol = beg → identification (search)
      □
      symbol = (var, x) → id_table ← x
      □
      symbol = (occur, x) → search (x)
      □
      symbol = ed → close id_table
     ]
   ]
  end identification ;
```

Neither "identification" nor search are exactly procedures because they can create activities (processes) which may survive their own termination. For this reason, we prefix them with **procp** which recalls **proc**edure and **p**rocess. The notation (**identifier** → **void**) describes the functional type of the **procp** "enclosing_block_search". Otherwise, we have used a CSP-like notation [11] to describe process behaviour.

Several comments may be made about this program :

1. It does not introduce superfluous data structures (only *vse id_table* is used).
2. It makes an intensive use of coordination structures : recursion for dealing with block nesting, higher-order functions for transfering a computation from

recursion level $n+1$ to recursion level n. This is necessary for achieving proper binding.

3. The indentifier table is processed without introducing any artificial sequentiality.
4. In order to detect missing declarations, the compiler provides the procedure *mission_declaration*, defined as :

```
proc missing_declaration = (identifier x) void :
  begin
    print (x, "cannot be identified")
  end missing_declaration ;
```

The initial call to identification is : *identification (missing_declaration)* ; If an identifier belonging to a block b cannot be bound to any declaration after invocation of search procedures associated with b and with all blocks enclosing b, the default procedure *missing_declaration* is invoked and produces the appropriate error message.

3 A smooth transition to Gamma

We believe that the above program is very elegant and clean ; it introduces as few data structures as possible and in particular, it avoids the use of intermediate data structures for emulating coordination structures.

However, dynamic sets of events are well adapted to the processing of very homogeneous sets because all the elements are associated with the same treatment. This may suggest that only a limited class of algorithms can be described with these structures. As far as software quality is concerned, one can realize that the provision of high-level data and coordination structures relieves the programmer from managing complex data structures. So, one can expect better quality programs.

Nevertheless, programs are seen as sets of processes which communicate and synchronize via events. It is recognized that reasoning and proving properties of such programs is very difficult, if not impossible, with the present tools. Then, our objective was to find more descriptive ways of computing over multisets. Further work was directed by the following observations :

1. The multiset is the less constraining data structure. There is no form of hierarchy in a multiset and there can be multiple occurrences of the same element.
2. A special ordering has been proposed on multisets [6]. Using such an ordering, it should be possible to prove termination properties.
3. In order to reason on multiset ordering, the computation should act as a multiset transformer.

Initial thinking let us to a proposal where a process was attached to every value of the multiset and a special value (weight) was associated to every such process [7]. Two processes could communicate if their respective weights were

related by a certain condition. The computation was considered as stable as soon as all conditions were false. Those familiar with Gamma will identify the chaotic behaviour of Gamma computations. This approach was still operational in its spirit, but there was a first step towards property proving (termination).

During some years, there was not much progress till discussions with Daniel Le Métayer lead to the present day Gamma formalism. Daniel was quite influenced by functional programming and in particular by Backus's work on FP languages [8]. The locality principle that is often put forward to qualify Gamma behaviour comes directly from the functional world. Furthermore, the functional approach was very precious for our objectives in formal property proving. So, the synthesis between parallel multiset processing and functional programming gave birth to Gamma which is sketched in the next section.

4 A short trip in the Gamma world

The purpose of this section is not a new survey of the work achieved on Gamma [5]. We simply want to convey the spirit of the language through a short selection of examples. The interested reader can find in [5] a more comprehensive account of the work done on Gamma during the last ten years.

4.1 A short introduction to Gamma

The Gamma formalism was proposed ten years ago to capture the intuition of a computation as a global evolution of a collection of atomic values "interacting" freely. Gamma can be introduced intuitively through the chemical reaction metaphor. The unique data structure is the multiset which can be seen as chemical solution. A program is simply a pair (Reaction condition, Action). Execution proceeds by replacing in the multiset elements satisfying the reaction condition by the results of the action. The final result is obtained when a stable state is reached, that is to say when no more reactions can take place. The maximum element problem of the introduction can be solved by the following Gamma program :

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

This program simply says that if two elements x, y are such that $x \leq y$, then these elements are replaced by y (the greater). Of course, nothing is specified about the order of evaluation of the comparisons. If several disjoint pairs of elements verify the condition, the corresponding reactions can be performed in parallel.

4.2 A selection of Gamma programs

We illustrate the Gamma style of programming through a set of selected examples, illustrating the elegance of the formalism.

Prime number generation

Goal : produce the prime numbers less than a given N.

Solution :
primes (N) = rem $\{2 \cdots N\}$
rem : $x, y \rightarrow y \Longleftarrow$ multiple (x, y)

Number sorting

Goal : sort a set of numbers, each number being represented by a pair (index, value)

Solution :
Sort : $(i, x), (j, y) \rightarrow (i, y), (j, x) \Longleftarrow (i > j) \wedge (x < y)$

Factorial

Goal : compute N !

Solution :
factorial (N) = fact $\{2 \cdots N\}$
fact : $x, y \rightarrow x * y \Longleftarrow$ **true**

The majority element problem

Goal : compute the majority element of a multiset M. This element appears more than card $(M)/2$ times in the multiset. For simplicity sake, we assume that such an element exists. The operation **one of** extracts a random element from a multiset.

Solution :
MAJ = **one of** maj (M)
maj : $x, y \rightarrow \{\} \Longleftarrow x \# y$

Convex hull

Goal : compute the smallest convext polygon containing a set of points in the plane

solution :
convex : $P_1, P_2, P_3, P_4 \rightarrow P_1, P_2, P_3 \Longleftarrow P_4$ **inside** $< P_1, P_2, P_3 >$
$Pi's$ are points and P_4 is "inside" $< P_1, P_2, P_3 >$ if P_4 is within the triangle they form.

The dining philosophers

Goal : solve the traditional dining philosopher problem.

Solution :

phil : $\varphi_i, \varphi_j \rightarrow \phi_i \Leftarrow j = i \oplus 1$

$\phi_i \rightarrow \varphi_i, \varphi_{i\oplus 1} \Leftarrow$ **true**

This solution contains two rules describing the two possible transitions : a thinking philosopher is allowed to eat (he/she gets two forks $\varphi_i, \varphi_{i\oplus 1}$) or an eating philosopher i starts thinking (he/she releases two forks $\varphi_i, \varphi_{1\oplus 1}$). $\oplus$ represents addition modulo 5, if we consider five philosophers.

This small set of examples demonstrates the richness and power of the Gamma paradigm. Many more examples are presented in [3].

4.3 Systematic program construction in Gamma

The most attractive properties of Gamma (high-level data structuring, locality principle) have been exploited in the design of a derivation method which can be applied to develop totally correct Gamma programs [4].

The derivation method is inspired by the work of Dijkstra [9] and Gries [10]. The method is composed of four stages. The first one is the transformation of the specification and its split into and invariant and a termination condition. In the second stage, the reaction condition is derived from the termination condition. The third stage is the deduction of the action from the invariant and the termination condition. The last stage is the derivation of a well-founded ordering from the action and the invariant for proving termination.

We will not review in detail this method, we will simply sketch the derivation of a sorting algorithm.

Specification

A natural data structure to describe a sorted set of values is the sequence. This sequence must be encoded within a multiset ; we choose a multiset M of pairs (index, value), where $x.i$ of an element x gives the position of the value $x.v$ in the sequence. Let M_o be the initial multiset ; a possible specification of the result M is :

(1) $\forall x, y \in M, x \cdot i < y \cdot i \Longrightarrow x \cdot v \leq y \cdot v$

(2) $M \cdot i = \{1 \cdot \cdot \text{ card } (M_0)\}$

(3) $M \cdot v = M_0$

1. **Split of the specification**

   We may choose $I = (2) \wedge (3)$ because (2) and (3) can be established in a straightforward way from M_0. I means that values are evenly distributed on the range $1 \cdot \cdot$ card (M_0). The termination condition T is (1) which must hold at the end of the computation.

2. **Reaction condition**

   The reaction condition can be derived in a straightforward way by negating T. So, we get :

$$R(x,y) = (x \cdot i < y \cdot i) \wedge (x \cdot v > y \cdot v)$$

3. **Action**

   The action must transform the multiset while maintaining the invariant. A simple reasoning shows that the only possible choice is :

$$A(x,y) = \{(x \cdot i, y \cdot v), (y \cdot i, x \cdot v)\}$$

   The elements $(x \cdot i, x \cdot v)$ and $(y \cdot i, y \cdot v)$ which are ill-ordered are replaced by $(x \cdot i, y \cdot v)$ and $(y \cdot i, x \cdot v)$ which are correctly ordered.

4. **Well-founded ordering**

   In order to prove the termination of a Gamma program, we have to provide a well-founded ordering on multisets and to show that the application of the action decreases the multiset according to this ordering. We use a result of [6] allowing the derivation of a well-founded ordering on multisets from a well-founded ordering on elements of the multiset. Let $>$ be an ordering on S and $\gg$ be the ordering on multisets $M(S)$ defined in the following way :

$$M \gg M' \Longleftrightarrow \exists X \in \mathcal{M}(S), \exists Y \in \mathcal{M}(S) \text{ such that}$$
$$(X \neq \{\} \wedge X \subseteq M \wedge M' = (M - X) + Y \wedge \forall y \in Y, \exists x \in X, x > y)$$

   The ordering $\gg$ on $\mathcal{M}(S)$ is well-founded if and only if the ordering $>$ on S is well-founded.
   In order to prove the termination of our *sort* program, we can use the following well-founded ordering [4] on the elements of the multiset :

$$(i,x) \sqsubseteq (i',x') \Longleftrightarrow i \geq i' \wedge x' \geq x$$

   This completes the derivation of the program presented earlier in section 4.2.

The derivation strategy which has been quickly developed here, has been applied successfully to a lot of non-trivial examples as described in [4]. We believe that this formal derivation process was made possible by the fact that, in Gamma, a program is no longer a sequence of instructions modifying a state, but rather a multiset transformer operating on all the data at once.

5 Conclusion

This paper describes a personal view of the genesis of Gamma. It all happened because we were searching for paradigms that were relying on very few high-level data structuring facilities which do not introduce any non-logical dependencies

between data. The multiset appears to be the ideal data structure. Then the question was : how to exploit multisets with language structures introducing no articial sequentiality and favoring formal reasoning. The outcome was this notion of dynamic set of events and its associate control structures which, after various influences, led to the Gamma formalism. We have stressed the simplicity and elegance of the Gamma formalism and of Gamma programs ; we have also shown how to develop Gamma programs from a specification in a systematic way. Much more could have been said (implementation, extensions $\cdots$), the interested reader find a review of past work and current perspectives on Gamma in [5].

Acknowledgments:

Many people have contributed to the work reported in this paper, most of them appear in the bibliography. I would like to mention two colleagues who had a major influence on this research : Laurent Trilling, now Professor at Grenoble University, who introduced me to the world of Programming and oriented some of my research directions ; Daniel Le Métayer, Research Director at Inria/Irisa for our common enthousiasm while discovering and developing the Gamma formalism.

References

[1] J.P. Banâtre, J.P. Routeau and L. Trilling. An event-driven compiling technique, *Communications of the ACM*, vol. 22–1, p. 34–42, january 1979.

[2] F. Andri, J.P. Banâtre and J.P. Routeau. A multiprocessing approach to compile-time symbol resolution, *ACM Transactions On Programming Languages And Systems*, vol. 3–1, p. 11–23, january 1981.

[3] J.P. Banâtre and D. Le Métayer. Programming by multiset transformation. *Communications of the ACM*, vol. 36–1, p. 98–111, january 1993.

[4] J.P. Banâtre and D. Le Métayer. The Gamma model and its discipline of programming. *Science of Computer Programming*, vol. 15, p. 55-77, 1990.

[5] J.P. Banâtre and D. Le Métayer. Gamma and the chemical reaction model : ten years after. *Proceedings of the Coordination'95 workshop*, IC-Press, London, 1996.

[6] N. Dershowitz and Z. Manna. Proving termination by multiset ordering. *Communications of the ACM*, vol. 22–8, p. 465–476, august 1979.

[7] J.P. Banâtre, M. Banâtre and P. Quinton. Constructing parallel programs and their termination proofs. *Proc. Int. Conf. on Parallel Processing.* Bellaire, USA, august 1982.

[8] J.Backus. Can programming be liberated from the Von Neumann style ? A functional style and its algebra of programs. *Communications of the ACM*, vol. 21–8, p. 613–641, august 1978.

[9] E.W. Dijkstra. *A discipline of programming.* Prentice Hall, Englewood Cliffs, NJ, 1976.

[10] D. Gries. *The science of programming.* Springer Verlag, New-York, 1981.

[11] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, vol. 21–8, p. 666–677, august 1978.

Graph Rewriting and Constraint Solving for Modelling Distributed Systems with Synchronization (Extended Abstract)

Ugo Montanari and Francesca Rossi

Università di Pisa, Dipartimento di Informatica
Corso Italia 40, 56125 Pisa, Italy
E-mail: {ugo,rossi}@di.unipi.it

Abstract. In this extended abstract we describe our approach to modelling the dynamics of distributed systems. For distributed systems we mean systems consisting of concurrent processes communicating via shared ports and posing certain synchronization requirements, via the ports, to the adjacent processes. We use graphs to represent states of such systems, and graph rewriting to represent their evolution. The kind of graph rewriting we use is based on simple context-free productions which are however combined by means of the synchronization mechanism. This allows for a good level of expressivity in the system without sacrifying full distribution. Moreover, to approach the problem of combining productions together, we suggest to exploit existing techniques for constraint solving. This is based on the observation that the combination problem can be modelled as a (finite domain) constraint problem. In this respect, we propose to use both local consistency techniques, to remove the possible redundancies in a system state, and a distributed backtracking search algorithm, as used in distributed constraint solving. Our method has two main advantages: first, it is completely formal and thus provides a precise description of the way a distributed system evolves; second, it also seems very promising from the performance point of view, since the techniques we propose to combine productions together have been proven very convenient in several cases.

1 Introduction

Among the many formalisms that can be chosen to represent distributed systems and their evolutions, we believe that graphs and graph grammars [Ehr78, Ehr87, SE94, SPvE92, CM95] are among the most convenient, both in terms of expressivity and of technical background. In fact, graphs describe in a natural way net topologies and data sharing, and moreover they possess a wide literature and technical results which make the whole field of graph rewriting very formal and its notions precisely definable. Therefore, consider a distributed system consisting of concurrent processes communicating via pieces of shared data (or channels, or ports). Then such a system can be represented as a graph