Electronic Notes in Theoretical Computer Science 50 No. 3 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume50.html 9 pages

Visualization of Distributed Algorithms Based on Graph Relabelling Systems [1]

M. Bauderon, S. Gruner, Y. Métivier, M. Mosbah, and A. Sellami [2]

LaBRI
Université Bordeaux 1 - ENSEIRB - IUT
(France)
{bauderon|metivier|mosbah|sellami}@labri.fr

Abstract

In this paper, we present a uniform approach to simulate and visualize distributed algorithms encoded by graph relabelling systems. In particular, we use the distributed applications of local relabelling rules to automatically display the execution of the whole distributed algorithm. We have developed a Java prototype tool for implementing and visualizing distributed algorithms. We illustrate the different aspects of our framework using various distributed algorithms including election and spanning trees.

1 Introduction

Visualization and animation of algorithms can assist in the design, in the debugging, in the validation and also in the explanation of algorithms [4,3]. Particularly, visualization may become extremely important for distributed algorithms because of the complexity of interprocess communication and synchronization [19]. In a distributed computation, events occur concurrently at many sites, and the state of each processor depends both on its internal actions and on messages received from other processes. Ability to display the exchange of messages and the current states of processes leads to intuition, to understanding and even to improving distributed algorithms. There is an important pedagogical interest associated with algorithm visualization, which can be used by students individually or in class demonstrations [28,33]

Extensive work has been done to integrate visualization to various phases of distributed computation [29], including design, analysis and implementation, and performance tuning and debugging (see [7,18,24,9,31]). Algorithm animation systems focus rather on the visualization of high level abstract events. There are many technical challenges raised by the animation of distributed algorithms. Conceptual frameworks are required to modularize and simplify the animation design process [27].

[1] This work has been supported by the European TMR research network GETGRATS, and by the "Conseil Régional d'Aquitaine".

[2] The current address of the second author is ECS, University of Southampton (England), sg@ecs.soton.ac.uk

We present in this work a method based on local graph transformations to visualize distributed algorithms. Our work goes beyond the known animation of isolated examples of distributed algorithms. We show that a large class of distributed algorithms, which can be described by some graph transformation systems, can be simulated and visualized automatically. Graph relabelling systems and, more generally, local computations in graphs are powerful models which provide general tools for encoding distributed algorithms, for proving their correctness and for understanding their power [14]. We consider an anonymous network of processors with arbitrary topology, represented as a connected, undirected graph where vertices denote processors, and edges denote direct communication links. An algorithm is encoded by means of local relabellings. Labels attached to vertices and edges are modified locally, that is on a subgraph of fixed radius k of the given graph, according to certain rules depending on the subgraph only ($k-$local computations). The relabelling is performed until no more transformation is possible. The corresponding configuration is said to be in normal form. Two sequential relabelling steps are said to be independent if they are applied on disjoint subgraphs. In this case they may be applied in any order or even concurrently.

The model of distributed computation is an asynchronous distributed network of processes which communicate by exchanging messages. To overcome the problem of certain nondeterministic distributed algorithms as well as to have efficient and easy implementations, we use randomization [6,32,25]. General considerations about randomized distributed algorithms may be found in [32] and some techniques used in the design and for the analysis of randomized algorithms are presented in [23,25,6]. Métivier et al. [20,21] have investigated randomized algorithms to implement distributed algorithms specified by local computations. Intuitively, each process tries at random to synchronize with one of its neighbours or with all of its neighbours depending on the model we choose, then once synchronized, local computations can be done. A synchronization between two neighbours is called a rendez-vous, and a synchronization between a vertex and all its neighbours is called a star synchronization. Procedures implementing synchronizations are given and discussed in [20,21]. We use these techniques to visualize the execution of a distributed algorithm. All random local synchronizations throughout the network are displayed, and messages exchanged during these synchronizations are also shown. Hence, the visualization of the execution of the whole algorithm is carried out until termination. We have developed a prototype tool with an interactive visual graph editor to build the network, and an interface to implement and visualize distributed algorithms.

The paper is organized as follows. Section 2 introduces graph relabelling systems, and their use to describe distributed algorithms. Section 3 presents a method to simulate and visualize distributed algorithms coded by graph relabelling systems. Section 4 presents future work and concludes the paper.

2 Graph Relabelling Systems

All graphs we consider are finite, undirected, simple and connected. A graph G is thus a pair $(V(G), E(G))$, where $V(G)$ is a finite set of vertices and $E(G) \subseteq \{\{v, v'\} \mid v, v' \in V(G),\ v' \neq v\}$ is the set of edges. Main notions may be found in [26].

An $L-$*labelled graph* is a graph whose vertices and edges are labelled with labels from a possibly infinite alphabet L. It will be denoted by (G, λ), where G is a graph and

$\lambda: V(G) \cup E(G) \to L$ is the *labelling function*. The graph G is called the *underlying graph* of (G, λ), and λ is a *labelling* of G. The class of $L-$labelled graphs will be denoted by $\mathcal{G}_L$, or simply $\mathcal{G}$ if the alphabet L is clear from the context.

Let (G, λ) and (G', λ') be two labelled graphs; (G, λ) is a subgraph of (G', λ'), denoted by $(G, \lambda) \subseteq (G', \lambda')$, if G is a subgraph of G' and λ is the restriction of λ' to $V(G) \cup E(G)$.

A mapping $\varphi: V(G) \cup E(G) \to V(G') \cup E(G')$ is a homomorphism of (G, λ) to (G', λ') if φ is a homomorphism of G to G' which preserves the labelling, that is such that $\lambda'(\varphi(x)) = \lambda(x)$ holds for every $x \in V(G) \cup E(G)$. An *occurrence* of (G, λ) in (G', λ') is an isomorphism φ between (G, λ) and some subgraph (H, η) of (G', λ').
In this paper, we only give an example and recall a few definitions of graph relabelling systems. For detailed results and various types of these systems, the reader should see [11,12,15,13,14].

Example: Distributed Computation of a Spanning Tree

Suppose that all the vertices are initially in some neutral state (encoded by label N) except exactly one vertex which is in an active state (encoded by label A) and that all edges have label 0.

At each step of the computation, an A-labelled vertex u may activate any of its neutral neighbours, say v. In that case, u keeps its label, v becomes A-labelled and the edge $\{u, v\}$ becomes 1-labelled.

Hence, several vertices may be active at the same time. Concurrent steps will be allowed provided that two such steps involve distinct vertices. The computation stops as soon as all the vertices have been activated. The spanning tree is given by the 1-labelled edges.

The algorithm may be encoded by the graph relabelling system $\mathcal{R}_1 = (L_1, I_1, P_1)$ defined by $L_1 = \{N, A, 0, 1\}$, $I_1 = \{N, A, 0\}$, and $P_1 = \{R\}$ where R is the following relabelling rule:

Figure 1 describes a sample computation using this algorithm. According to the previous discussion, the reader should keep in mind that some of the relabelling steps *may* be applied concurrently.

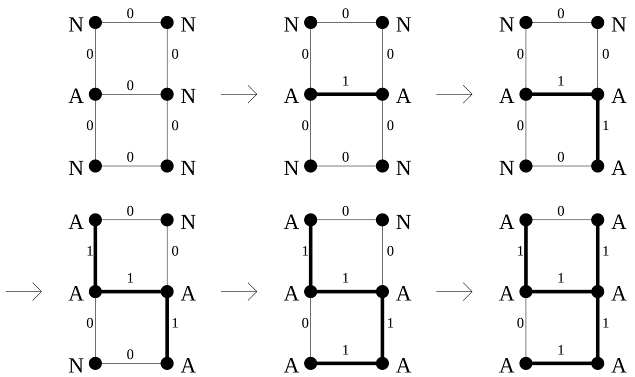


Fig. 1. Distributed computation of a spanning tree

Note that other relabelling systems, which have different behaviour with respect to termination, can be used to generate spanning trees (see [13,2]).

Graph relabelling systems and more generally local computations satisfy the following constraints which seem to be natural when describing distributed computations with a decentralized control:

($C1$) they do not change the underlying graph but only the labelling of its components (edges and/or vertices), the final labelling being the result of the computation,

($C2$) they are *local*, that is, each relabelling step changes only a connected subgraph of a fixed size in the underlying graph,

($C3$) they are *locally generated*, that is, the application condition of the relabelling only depends on the *local context* of the relabelled subgraph.

For such systems, the distributed aspect comes from the fact that several relabelling steps can be performed simultaneously on "far enough" subgraphs, giving the same result as a sequential realisation of them, in any order.

Graph relabelling system are defined by a finite set L of labels (labels used in the relabelled graphs), a set $I \subseteq L$ of initial labels (every graph starting a relabelling process has only labels in I) and a finite set of relabelling rules ; it may be equipped with a mechanism which locally controls the application of the relabelling rules e.g. priority, forbidden contexts. A relabelling rule r consists of the relabelling of a fixed connected subgraph G_r :

$$r : (G_r, \lambda) \longrightarrow (G_r, \lambda')$$

We say that a labelled graph (G, l) is relabelled into (G, l') if there exists a finite sequence of *allowed applications* of relabellings leading from (G, l) to (G, l').

Among the classical distributed algorithms, which can be encoded by graph relabelling systems, we recall the following [2]:

- Distributed computation of a spanning tree with local detection of the global termination [13]
- Election in trees, and in complete graphs [15]
- Mazurkiewicz's universal graph reconstruction algorithm [17]
- Detection of stable properties (Szymanski, Shi and Prywes [30]).

3 Deriving Visualization of Distributed Algorithms

Consider a graph representing a network, where nodes correspond to processors and edges correspond to communication channels. The visualization of a distributed algorithm consists of showing and animating its execution. Data exchanged between processors, as well as status and label updates of processors and of channels are displayed on-the-fly on the screen. Of course, other interesting events depend on the algorithm itself. For instance, to determine a spanning tree, it is important to mark edges belonging to the spanning tree.

The task of animating a distributed algorithm in our approach relies mainly on the choice of a type of local computations, and on the design of a relabelling system. The former defines the model of local computations performed by the rules of the relabelling system.

3.1 Types of Local Computations

There are three types of local computations as investigated in [20,21]. Implementation of these local computations for an aynchronous message passing system needs randomized procedures [2]. For the purpose of visualization, this randomized implementation is useful because it enables the end-user to observe the entire execution of the algorithm. These local computations are:

- *Rendez-vous* (RV): in a computation step, the labels attached to vertices of K_2 (the complete graph with 2 vertices) are modified according to some rules depending on the labels appearing on K_2. To implement RV, we consider the following distributed randomized procedure. The implementation is partitioned into rounds; in each round each vertex v selects one of its neighbours $c(v)$ at random. There is a rendezvous between v and $c(v)$ if $c(v) = v$, we say that v and $c(v)$ are synchronized. When v and $c(v)$ are synchronized there is an exchange of messages by v and $c(v)$. This exchange allows the two nodes to change their labels.
- *Local Computation 1* (LC1): in a computation step, the label attached to the center of a star is modified according to some rules depending on the labels of the star, labels of the leaves are not modified. The implementation of LC1 is the following randomized local election. it is partitioned into rounds, and in each round, every processor v selects an integer $rand(v)$ randomly from the set $\{1, ..., N\}$. The processor v sends to its neighbours the value $rand(v)$. The vertex v is elected in the star centered on d and denoted S_v, if for each leave w of S_v : $rand(v) > rand(w)$. In this case a computation step on S_v is allowed : the center collects labels of the leaves and then changes its label.
- *Local Computation 2* (LC2): in a computation step, labels attached to the center and to the leaves of a star may be modified according to some rules depending on the labels of the star. The implementation of LC2 is the following randomized local election. it is partitioned into rounds, and in each round, every processor v selects an integer $rand(v)$ randomly from the set $\{1, ..., N\}$. The processor v sends to its neighbours the value $rand(v)$. When it has received from each neighbour an integer, it sends to each neighbour w the max of the set of integers it has received from neighbours different from w. The vertex v center of the star S_v is elected if $rand(v)$ is strictly greater than $rand(w)$ for any vertex w of the ball centered on v of radius 2; In this case a computation step may be done on S_v. During this computation step there is a total exchange of labels by nodes of S_v, this exchange allows nodes of S_v to change their labels.

3.2 Implementation of Relabelling Systems

We will refer to the previous types of local computations by *synchronization*. Now, we will show how to combine synchronization and relabelling rules, in such a way that a relabelling system can be applied randomly on the network. Each processor tries to get a synchronization with one of its neighbours, or with all its neighbours, depending on the type of local computations discussed above. Once, a processor v is involved in a synchronization, a rewriting step can be performed. That is, v exchanges labels and attributes with its neighbour(s), checks if a left-hand side of one of the rules is found (w.r.t isomorphism), and if so, updates its labels and its attributes according to the righthand side of the rule. Then, the synchronization is broken, and v and its neighbour(s) are free to re-try new synchronizations. Note that the relabelling rules required for all our examples are either K_2 rules or star rules.

3.3 ViSiDiA: A tool for Visualizing Distributed Algorithms

We have developed a prototype tool called ViSiDiA [1,2] to help to implement and visualize relabelling systems as described above. As it is written in the Java language, the processors are simulated by Java threads. To program a relabelling system, a library of high level primitives allows the user to implement local computations. In particular, three functions (*rendezVous(), starSynchro1() and starSynchro2()*) implementing the previous synchronizations are provided. Moreover, communications between processors can be expressed by primitives such as *sendTo(neighbour, message)*, and *receiveFrom(neighbour)*. An illustrative example shows the implementation of the spanning tree example discussed in Section 2.

To visualize a relabelling system, the end-user must first create a graph modelling the

Algorithm 1 Implementation of Spanning tree

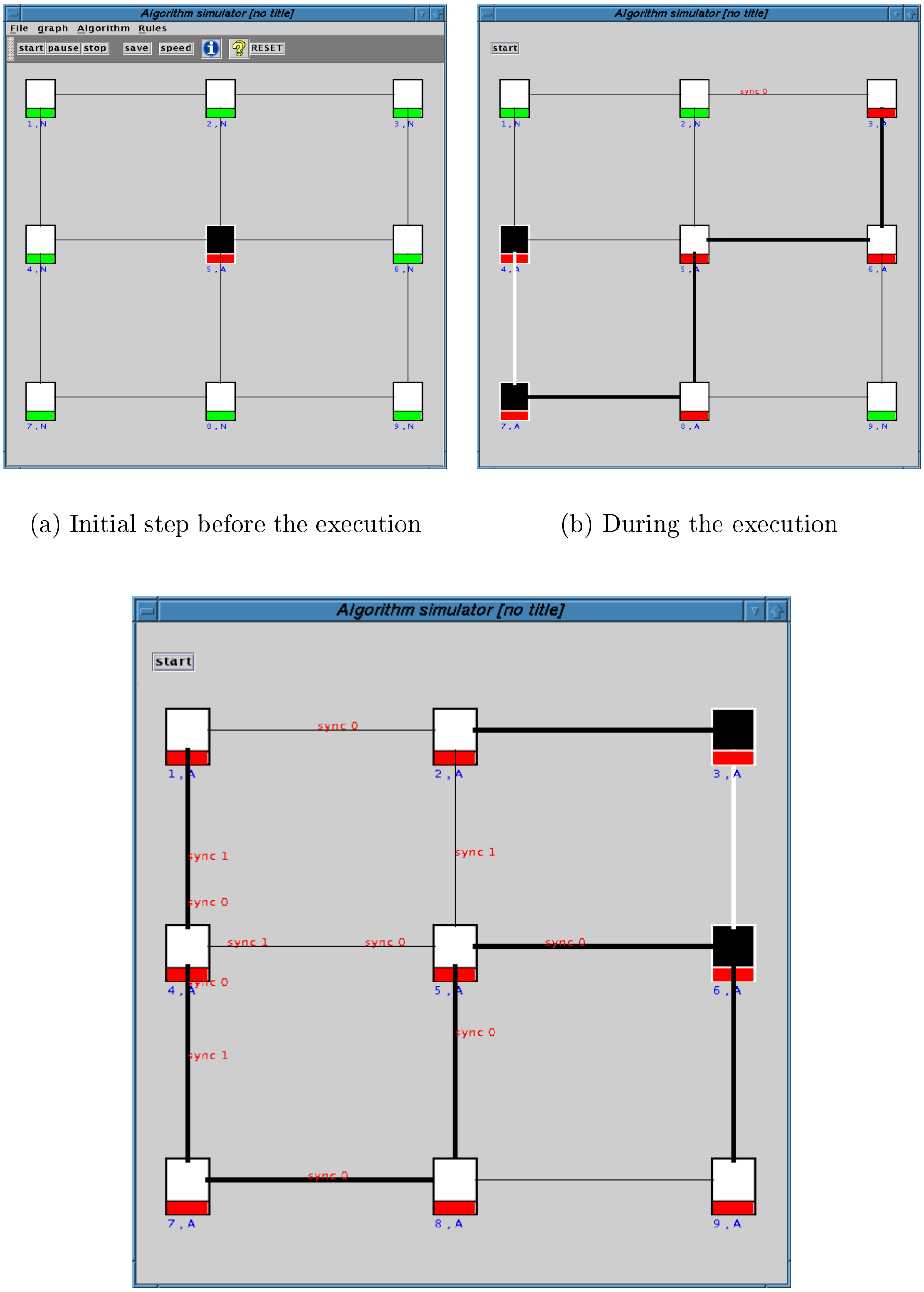
```
while (run) {
   neighbour = rendezVous();
   sendTo(neighbour,myLabel);
   neighbourLabel=receiveFrom(neighbour);
   if (myLabel == 'N') && (neighbourLabel == 'A'){
        myLabel = 'A';
        edge[neighbour]=1
        }
   breakSynchro();
}
```

network. To do so, our tool has a friendly Graphical User Interface to construct an arbitrary graph using the buttons of the mouse (See Fig. 2(a)). The visual attributes of a node (labels, colors, shapes) can be customized by the user.

A control panel allows the user to play animation, pause it at any point during its execution, and stop it. The user can also choose a node and set its label to A. For the example, the label of node 5 is A. To start the animation, the user presses the start button. In this case, ViSiDiA creates automatically a thread for each vertex. Fig. 2(b) shows the state of the network during the animation. In this snapshot, nodes 4 and 7 have a rendez-vous synchronization. All edges where the relabelling system rule has been applied belong to the spanning tree. Finally, Fig. 2(c) shows the resulting spanning tree. The correctness of this relabelling system is proved in [13].

Many distributed algorithms are already implemented and can be directly animated [2]. These include the following

- leader election in trees, in chordal graphs and in complete graphs,
- randomized rendez-vous and randomized local elections,
- spanning tree,
- Mazurkiewicz's universal graph reconstruction,
- detection of stable properties



(a) Initial step before the execution

(b) During the execution

(c) Result of the execution

Fig. 2. Visualization of the computation of a spanning tree

An interesting advantage of our approach is that we only need to implement local relabellings to code complicated distributed algorithms. Therefore, visualizing the execution of these algorithms consists of animating distributed local computations. Moreover, our implementation preserves the properties of relabelling systems such as correctness and termination.

4 Conclusion

In this paper, we have briefly presented the current state of our work on the visualization of distributed algorithms based on graph relabelling systems. We think that graph transformations are useful to simplify and describe in a uniform and efficient way distributed applications [8,22,31]. However, work remains to improve our tool particularly to handle huge graphs, and also real networks. Parts of the tool are under development with the goal of providing more intuitive interactions and displays. We have used our tool to make many experiments useful for the analysis of several distributed algorithms [1,2]. We think that our tool is useful for pedagogical purposes to explain the execution of distributed algorithms, and also for researchers in distributed algorithms who require tools for tests and experiments.

References

[1] M. Bauderon, S. Gruner and M. Mosbah: A New Tool for the Simulation and Visualization of Distributed Algorithms. LaBRI Report 1245-00, Université Bordeaux 1, October 2000. (Accepted in MFI'01, Toulouse, 21-23 May 2001)

[2] M. Bauderon, Y. Métivier, M. Mosbah and A. Sellami: From local computations to asynchronous message passing systems. LaBRI Report, Université Bordeaux 1, 2001. In preparation.

[3] M. Brown. Zeus: A System for Algorithm Animation and Multi-View Editing. *VL'91 Proceedings in Visual Languages*, pp.4-9, Oct.1991

[4] M. Brown. *Algorithm Animation.* MIT Press, 1988

[5] H. Ehrig, H.-J. Kreowski, U. Montanari and G. Rozenberg (Eds.): *Concurrency, Parallelism and Distribution — Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3.* World Scientific Publ. Singapore 1999

[6] R. Gupta, S. A. Smolka and S. Bhaskar. On randomization in sequential and distributed algorithms. *ACM Comput. sur.*, 26(1):7–86, 1994.

[7] M.T. Heath, AD. Malony and D.R. Rover. The visual display of parallel performance data. *IEEE COmputer,* 28(11), 21–28, 1995.

[8] D. Janssens. *Actor Grammars and Local Actions.* Chpt.2, pp.57-106 in [5]

[9] J. Joyce, G. Lomow, K. Slind and B. Unger. Monitoring distributed systems *ACM Transactions on Computer Systems,* 5(2):121–150,1987.

[10] M. Leuschel. *Design and Implementation of the High-Level Specification Language CSP(LP).* Proceedings of PADL'01, LNCS, Springer-Verlag Berlin 2001

[11] I. Litovsky and Y. Métivier. Computing trees with graph rewriting systems with priorities. *Tree automata and languages*, pages 115–139, 1992.

[12] I. Litovsky and Y. Métivier. Computing with graph rewriting systems with priorities. *Theoret. Comput. Sci.*, 115:191–224, 1993.

[13] I. Litovsky, Y. Métivier and E. Sopena. Different local controls for graph relabelling systems. *Math. Syst. Theory*, 28:41–65, 1995.

[14] I. Litovsky, Y. Métivier and E. Sopena. Graph relabelling systems and distributed algorithms. In H. Ehrig, H.J. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of graph grammars and computing by graph transformation*, volume 3, pages 1–56. World Scientific, 1999.

[15] I. Litovsky, Y. Métivier and W. Zielonka. On the recognition of families of graphs with local computations. *Information and Computation*, 118(1):110–119, 1995.

[16] N. Lynch. A hundred impossibility proofs for distributed computing. In *8th International Conference on Distributed Computing Systems*, pages 1–28, 1989.

[17] A. Mazurkiewicz. Distributed enumeration. *Inf. Processing Letters*, 61:233–239, 1997.

[18] C.E. McDowell and D.P Helmbold. Debugging concurrent programs. *ACM Comput. Surv.,* 21(4):593–622, 1989.

[19] A. Mester and H. Krumm. Animation of Protocols and Distributed Systems. *Journal of Computer Science Education,* 10(3):243–265, 2000.

[20] Y. Métivier, N. Saheb and A. Zemmari. Randomized rendezvous. In *Mathematics and computer science : Algorithms, trees, combinatorics and probabilities*, Trends in mathematics, pages 183–194, 2000.

[21] Y. Métivier, N. Saheb and A. Zemmari. Randomized local elections : probabilistic and efficiency analysis. Technical report, LaBRI, university of Bordeaux1.

[22] U. Montanari, M. Pistore and F. Rossi: Modeling Concurrent, Mobile and Coordinated Systems via Graph Transformation. Chpt.4, pp.189-268 in [5]

[23] R. Motwani and P. Raghavan. *Randomized algorithms.* Cambridge University Press, 1995.

[24] C.M. Pancake and S. Utter. Models in parallel debuggers. *Proceedings of Supercomputing'89*, 627–636, 1989.

[25] J. Ramirez-Alfonsin, M. Habib, C. McDiarmid and B. Reed, editors. *Probabilistic Methods for Algorithmic Discrete Mathematic.* Springer-Verlag, 1998.

[26] K.H. Rosen, editor. *Handbook of discrete and combinatorial mathematics.* CRC Press, 2000.

[27] J.T. Stasko. The path-transition paradigm: a practical methodology for adding animation to program interfaces. *J. of Visual Languages and Computing,* 1(3):213–236, 1990.

[28] J.T. Stasko, A. Badre and C. Lewis. Do algorithm animations assist learning ? an empirical study and analysis. In *Proc. ACM Conf. Human Factors in Computing Systems,* pages 61–66, 1993.

[29] J.T. Stasko and E. Kraemer. *The Visualization of Parallel Systems — An Overview.* Journal of Parallel and Distributed Computing, Vol.18 No.2, pp.105-117, 1993

[30] B. Szymanski, Y. Shi and N. Prywes. Terminating iterative solutions of simultaneous equations in distributed message passing systems. In *4th International Conference on Distributed Computing Systems*, pages 287–292, 1985.

[31] G. Taentzer, I. Fischer, M. Koch and V. Volle: Distributed Graph Transformation with Application to Visual Design of Distributed Systems. Chpt.5, pp.269-340 in [5].

[32] G. Tel. *Introduction to distributed algorithms.* Cambridge University Press, 2000.

[33] A. Van Dam. The electronic classroom: Workstations for teaching. *Int. J. of Man Machine Studies?* 21(4):353–363, 1984.