A Tool for Interactive Visualization of Distributed Algorithms *

Stefan Gruner[2], Mohamed Mosbah[1], and Michel Bauderon[1]

[1] LaBRI
ENSERB / Université Bordeaux 1
351 Cours de la Libération
33405 Talence Cdx, France
{mosbah|bauderon}@labri.u-bordeaux.fr

[2] Dept. of Electronics and Computer Science
University of Southampton
Highfield
Southampton SO17 1BJ, England
sg@ecs.soton.ac.uk

Abstract. We present a tool for the visualization of distributed computations. Special attention is payed to certain distributed algorithms which have been coded as rewriting systems. In order to study the behaviour of algorithms and tool, several experiments have been performed the results of which are presented and discussed. Finally, some important properties of our tool are described and explained. A release of the tool is available to the public.

1 Introduction

In recent years, distributed and parallel systems [1, 19] are generally available, and their technology has reached a certain degree of maturity. Moreover, the technological advances of workstations and networks combined with middleware solutions like CORBA and DCOM help to avoid physical hardware problems such as low speed, heterogeneity, failure, etc. Thus, distributed systems have become a serious option for many companies. Examples of distributed applications include manufacturing, banking, process control, or weather forecast computations.

Unfortunately, we still lack complete understanding of how to design, realize, and test the software of such systems, although research effort has been spent on this topic. Understanding the behaviour of a distributed program remains a challenge. This is due to the intrinsic complexity of distributed algorithms and programs compared to serial ones. Programmers must coordinate and synchronize communication between processes. Several tools have been proposed to

* This work has been supported by the TMR research network GETGRATS [4] and by the "Conseil Régional d'Aquitaine"

assist the development of distributed applications. These tools include debuggers, performance monitoring, execution analysis, and other aids.

Our work has focus on a *tool for visualizing and animating* distributed algorithms [5]. A key component of understanding a distributed computation is knowing what is occurring in the algorithm, how processors are working and how they communicate. Since distributed algorithms are complex abstract objects, visualization plays an important role in improving their perception and their understanding. Researchers in distributed computing need visualization during the design and debugging phase of an algorithm and also when presenting the algorithm. This is especially important in case of certain graph problems which are know to be deterministically unsolvable by distributed computation [9] such that random-based approaches to solve such problems are inevitable: obviously, due to the essence of randomization, the actual runtime behaviour of such random-based computations can only be observed but not theoretically predicted in principle. In the classroom, teachers can present distributed algorithms and use visualization to explain the execution steps in detail. Moreover, students are able to manipulate and experiment existing distributed algorithms through visualization, and they can see their results on selected input as well as the behaviour of processors during the execution.

There are some special visualization systems developed for the needs of particular projects, and general systems for visualizing large families of algorithms. Examples of general systems include Balsa [2], Polka [16], Zeus [3] and Xtango [14]. The tool we have developed deals with many distributed algorithms. Moreover, we suggest to the user, through easy and friendly-user interface, to describe the algorithm to visualize using rewriting rules according to the methods described by Métivier et al. [8, 9, 10, 11].

The rest of our paper is structured as follows. Section 2 sketches the underlying model which our tool is based on. The tool itself is described in section 3. Section 4 briefly reports on our experience with the behaviour of our tool in the execution of distributed algorithms. Related work is described in section 5, and section 6 concludes the paper.

2 Overview of the underlying Model

In this section, our notion of *distributed computation* is (informally) explained, which is necessary to understand the tool description of section 3. We sketch the structures in which distributed computation takes place as well as the entities of which those structures consist, and we describe what temporal and causal constraints have to be obeyed by the visualisation component of our prototype. Our model is *application-transparent* which means that it does not matter if we plug our visualisation tool into a real network (which is ongoing work) or if we internally simulate such a network in other modules of our software prototype (which is done in its currently available version).

2.1 Autonomous Distributed Computation

In *autonomous* distributed computation, which our concept is based on, we have neither a master nor *one* program which is shared among the processors. Instead we have *as many programs as processors* —the programs may be clones of each other but this need not be the case— and the processors have to organize their cooperation themselves.[3]

One can distinguish several different sub-types of autonomous distributed computing, for example: Does each processor have knowledge about the whole system or does each processor only know its immediate neighbour processors? Do the processors carry unique identifiers or not?

In our approach we assume *anonymous* distributed computing which means that each processor does only know its immediate neighbours, and the processors do not carry unique identifiers.

2.2 Nets and Graphs

Having sketched the concept of autonomous distributed computing, which is the principle of our approach, we now have to explain the substrate on which this kind of computing shall take place. In general, such substrates are *nets* of arbitrary topological structure which we represent by undirected *graphs* consisting of nodes and edges as usual.

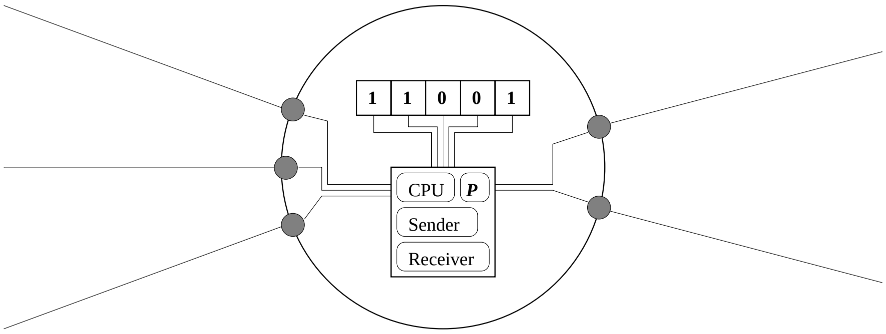

Fig. 1. *processor with program P, current state* $s = 11001$*, and* $C = 5$

[3] The well-known *Neural networks* and *cellular automata* can also be seen as special cases of autonomous distributed systems. In case of autonomous distributed computing, the notion of algorithm is not the classical one any more. This is because the behaviour of the whole distributed system is not sufficiently described by each occurring algorithm executed by its corresponding processor. There is no meta algorithm explicitly describing the system behaviour emerging from the cooperation of the autonomous processors or entities. Such systems are also called *computationally irreducible.*

Nodes: We assume that each node in the graph represents an entity called *processor* which is able to Turing-compute and which is able to send and receive *messages* to its neighbour processors.

We further assume that the internal *state space* S_k of a node k is related to its *connectivity* $C = |E_k|$ (which is the number of edges to which this node is attached in the graph) as $S_k :\supseteq \{0,1\}^C$. This means that k at least keeps a memory cell s_i for each of its immediate neighbour nodes n_i $(i = 1, \ldots, E_k)$. If k has established a certain relationship with n_i then $s_i := 1$, otherwise $s_i := 0$.

A simple example is sketched in **Fig.1**. (Of course, a node's current state $s \in S_k$ is not necessarily invariant: usually it will change from time to time during an ongoing distributed computation.)

Edges: The edges of a graph are meant to represent communication *channels* under the following (ideal) assumptions.

- The channels are bi-directional.
- Messages travelling through a channel cannot overtake each other while travelling, which means that the channels operate in a *FIFO* mode.
- The channels are *perfect* such that no message can be lost (or changed) while travelling through a channel.
- Finally we assume *full duplex* communication, which means that two messages can pass each other while traveling in opposite directions at the same time, without disturbing each other.

With these assumptions we do not need to model the channel as an active entity with a semaphore blocking technique. (The channels are thus completely passive in our model.) The situation is sketched in **Fig.2**.

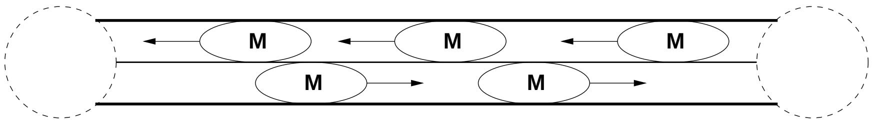

Fig. 2. *full duplex FIFO channel with travelling messages*

Please note that our decision to model processors as nodes and channels as edges is not as "natural" as it might appear. In [12] for example we find distributed systems modelled with processors as edges and channels as nodes. The purpose of [12], however, is rather different from ours: They are dealing with formal semantics of concurrent systems, while we are on the way to intuitive online simulation and visualisation tools, as outlined in the introductory section.

Communication: In our model we assume *asynchronous* communication. Thereby, some processor p_i may seamlessly continue its operations after it has sent some message m to a neighbour processor p_j via a communication channel e.[4]

2.3 Partially ordered Time

Dealing with visualisation of distributed systems, one always has to face the problem of how to sequentially display a set of *events* on a screen which have possibly taken place simultaneously in a distributed computation network. This is a matter of *causality* and *time* which has somehow to be reflected by the underlying model of event simulation and visualisation.

In order to keep our model as general and as reliable as possible, we must not impose too many constraints on it (which could turn out as not satisfiable in practice). The following assumptions, however, seem to be the weakest possible.

No Common Clock: Assuming anonymous networks where each node does only know its immediate neighbours (as described above), having a common clock would mean having a graph with at least one (clock)node connected to all other nodes (as Information can only be transmitted via the edges). On the other hand, we cannot impose any constraint on the topological shape of our graphs a-priori. The visualisation component of the tool cannot serve as a common clock either, because it must not artificially influence the net processes which it has just to display on the screen.

Communication Speed Unknown: Let S_I and S_T be the initial state and the terminal state of a distributed computation. Let M be the set of all messages which occurred in this computation, thus: $S_I \xrightarrow{M} S_T$. Let $m \in M$ any message and $v(m)$ its velocity (travelling speed). As $v(m)$ is dependent from the hardware realization of the distributed system, which is not characterized by our model, we may only assume $0 < v(m)$. For the same reason, we may not even assume an equality $v(m_i) = v(m_j)$ for two different messages m_i and m_j in M.

Network Geometry Unknown: The graphs which we use to characterize the topological properties of our communication networks do not contain any information about the length of the communication channels — and length is a geometrical dimension. Those channel measures are dependent from the hardware realization (wires) of the system which is not specified by the model.

A consequence of these assumptions is that each node p_i in our network has got its *private time line* which is only weakly related to the time lines of the other nodes in the net, and a total order of events in such a network is impossible from a local point of view [6].

These minimal properties of our model make it difficult for the visualisation component of our prototype to display a logically consistent "movie" of the state changes during a distributed computation. In [5] we have described two

[4] With synchronous communication, p_i would have to stop all its operations after sending m until it would have received an answer *reply*(m) by p_j via e.

concepts (one of which based on Lamport's clocks [6]) for achieving the necessary consistency of visualisation.

3 Implementation

We have implemented a prototype tool based on the concepts discussed in this paper. The user interface of the tool is a graphical environment that allows the user to draw a network easily, and to visualize the execution of a distributed algorithm. The architecture of the tool is composed of mainly three parts: the graphical user interface, the simulator and the algorithm library as sketched in **Fig.3**. These modules are well-separated such that the modification of one component involves only small changes for the rest of the system.

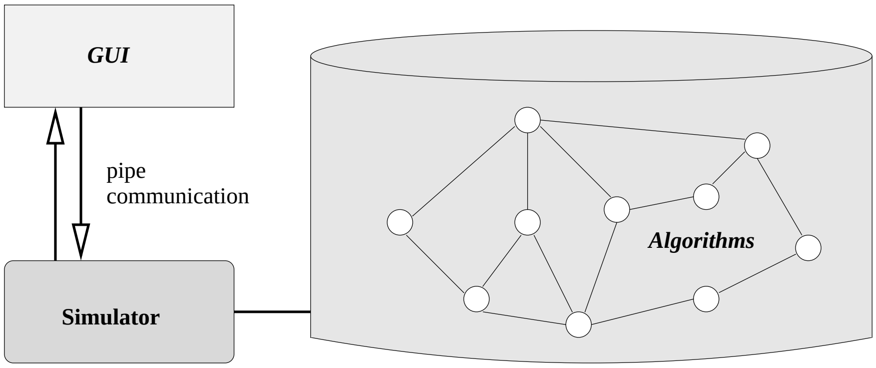

Fig. 3. *architecture of the prototype*

3.1 The Graphical User Interface

The editor allows the user to construct a network by "Drag & Drop". The user can add, delete, or select vertices, edges or subgraphs. The visual attributes of a vertex —labels, colors, and shapes— can also be set by the user. Once the network drawn, the simulation is run after the user has chosen an algorithm or a rewriting system from the library. During the execution, the traffic of messages exchanged between nodes and their values are displayed and the status of edges and nodes are updated on-line. Moreover, the visualization speed can be chosen by the user such that the messages can be seen travelling slowly or fast, depending on the purpose of the current application. **Fig.4** shows the initial state of the distributed algorithm that generates a random spanning tree as described in [11]. The result of the execution of the algorithm is shown in **Fig.5**. Detailed descriptions of the tool are given in [5].

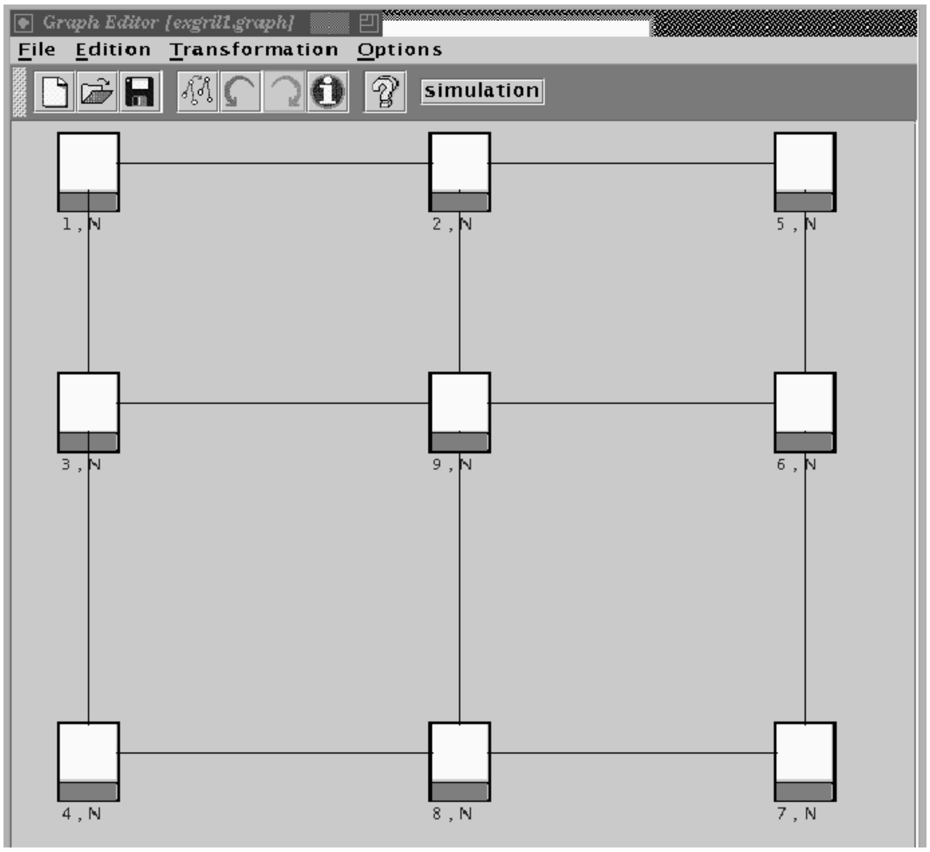

Fig. 4. *before the distributed spanning tree computation*

3.2 The Simulator

The simulator is the link between the visualizer and the algorithms. It models a network of asynchronous processors. Each processor communicates only with its immediate neighbours by message passing. In the current version of the tool, each processor, associated to a vertex in the underlying graph, is implemented by a JAVA *thread*. The simulator manages the exchanges of messages between threads, as well as the visualization of events. Each event has an identifier which is a number that will be used to synchronize the execution of the algorithm with its visualization. The identifier of an event helps to acknowledge its visualization. In this way, displayed information is synchronized with the state of the network. There are two types of events corresponding to the state modification of vertices or edges, or to message exchanges:

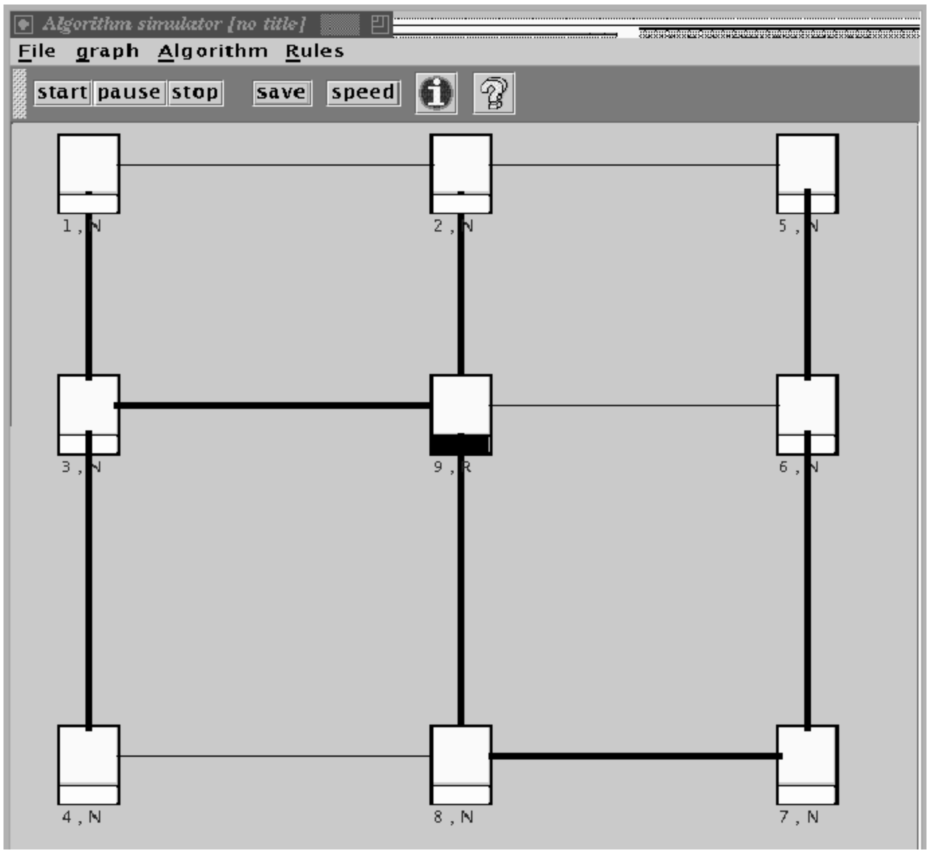

Fig. 5. *after the distributed spanning tree computation*

– **State modification** of a vertex: The state of a vertex is a set of information of the vertex which are used by the algorithm during its execution. These informations are interpreted by the graphical interface. Precisely, when a vertex changes its state, it informs the simulator who sends an event to the graphical interface. The process of this vertex waits until obtaining an acknowledgment receipt from the graphical interface, which means that this event has been displayed. The changes of an edge are displayed in a similar way.

– **Message visualization**: Processors use queues to store messages. Sending a message M from vertex A to B consists of adding M to the queue of B. However, the visualization of the message M moving from A to B is displayed before adding it effectively to the queue of B. The simulator begins by sending an event to the graphical interface in order to visualize the message M. Once this event is acknowledged by the simulator, it is added to the queue of B, and a signal is sent to the processor of B in order to wake it in case it had been blocked on reading an empty queue.

3.3 The Algorithm Library

An algorithm is implemented by a JAVA program which will be instantiated on each vertex of the graph, and executed asynchronously by the corresponding processor. A vertex is implemented by a class that contains its identifier, its internal state, its degree, and optionally the size of the graph. It is possible for the programmer to manipulate a vertex by using the following implemented interface functions:

- `getId()` returns the identity of a vertex,
- `getState()` (resp. `setState()`) gives (resp. changes) the state of a vertex,
- `getArity()` returns the degree of a vertex, i.e. the number of neighbours,
- `getNetSize()` allows to know the size of the graph for algorithms where the size is assumed to be known.

Since communications between processors are based on messages, the required functions to handle messages are provided. A message is programmed by a class that contains all required informations. The programmer can use a message through the following methods:

- `sendTo(int)` (resp. `sendAll()`) sends the message to a particular neighbour (resp. all neighbours),
- `receiveFrom(int)` (resp. `receive()`) receives a message from a particular neighbour (resp. the first message waiting in the queue of the vertex). Other methods are also available to manage messages of particular types, i.e. messages that send integer, strings, or other types.
  Remember that although messages are stored in the queue of the receiver, the implementation allows also to handle messages that arrive on a particular channel.

To animate *random* (or probabilistic) distributed algorithms, a method called `synchronization()` is implemented. It returns a random synchronization of a vertex with one of its neighbours. To do so, every vertex sends to its neighbours either 0 or 1 (the choice is random). Two vertices are synchronized if they send each other the value 1. For more details on the synchronization and its use in random distributed algorithms, see [10].

Many standard distributed algorithms are already implemented. These include the *leader election* and *star synchronization* [9] as well as *spanning tree computation* [11]. The tool provides also a graphical interface to capture a *rewriting system* as is known that rewriting systems can be used to describe distributed algorithms [8]. The advantage of using rewriting systems is that the algorithm can be defined by a small number of production *rules*. A production rule is implemented by a class called `AlgoRule`.

4 Experiments with the Tool Prototype

Having described the architecture of our prototype in the previous section, we can now report on some behavioural properties of it. The results of our experiments will certainly influence later versions and forthcoming implementations of our visualization tool. They may be regarded as first steps into further studies.

When performing experiments with the tool one has to keep in mind that *tool and algorithm melt into a unit* the particular constituents of which are hard to separate. This tight integration between tool and algorithm is not easy to handle in an empirical study, because:

- When we want to exactly explore the behaviour of the tool, we need an algorithm as "driving fuel" the behaviour of which must not be unknown.
- When we want to exactly explore the behaviour of an algorithm, we need a tool as "vehicle" the behaviour of which must not be unknown.

The problem of our experimental situation is, however, that we *neither* know the behaviour of our new prototype exactly, *nor* the behaviour of the algorithms which have not yet been subject to empirical investigations. For this reason the reader has to be aware of the possibility that the results presented in this section may still be somewhat tentative. Nevertheless we hope to sufficiently approximate the "unknown reality".

We have performed several experiments with our tool the results of which are described and interpreted in [5]. They cannot be presented in this paper due to lack of space, but one example is given in the following.

Example: *Ring graphs* with even numbers of nodes are simple examples of *non*-T-prime graphs [11]. For this reason Métivier's distributed SpanningTree algorithm [11] —which has also been employed for screen-shots in section 3— sometimes fails when applied to such graphs, because SpanningTree is explicitly designed to operate on T-prime graphs. For this experiment, we have made 237 repetitions of SpanningTree on a 6-Ring and observed the distribution of results with respect to failure or success [5]. Because our ring graph has size 6, we can have failure outputs with 2 part-trees of size 3 or with 3 part-trees of size 2. **Fig.6** shows our ring graph with one example of a successful output and two examples of failures. All in all we have 36 success possibilities wherein a spanning tree is constructed. In total we have 79 output possibilities such that 43 of them are failures. The combinatorial success-failure-ratio is thus $(36 : 43) \approx (9 : 11)$.

As depicted in **Fig.7**, only 43 of all the 79 output possibilities have been observed at all. Success has been noticed in 172 cases, while failure was counted 65 times. The six dark grey "towers" in the success part of Fig.7 show that the 6 nodes are almost equally returned as head-nodes of the computed spanning tree; this might be regarded as a consequence of the perfectly symmetrical structure of ring graphs which equal connectivity for each of its nodes. While the combinatorial success-failure-quota is only $(9 : 11)$, the empirically observed success-failure-quota of $(172 : 65) \approx (3 : 1)$ is considerably better, which was not obvious before the experiment had been performed.

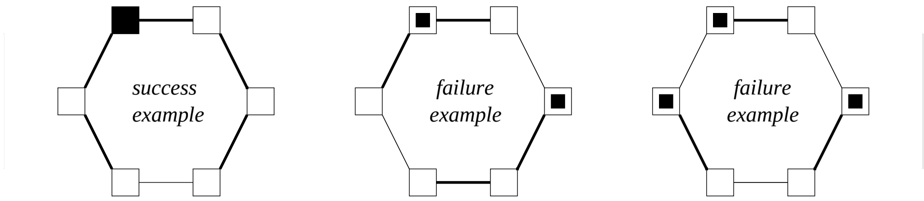

Fig. 6. *ring graph with example outputs of success and failure*

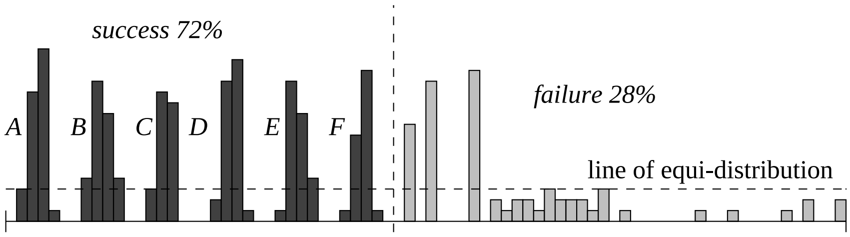

Fig. 7. *distribution of outputs in the ring graph experiment*

5 Related Work

There are other related tools available to assist the users in comprehending the execution of algorithms. Those tools generally perform some kinds of behavioural analysis to help convey the user exactly how the programs operate. For a summary of algorithm animation and software visualization systems see [13].

- PARADE [17] is an environment for developing visualizations and animations of parallel and distributed programs. PARADE handles trace events and monitoring. It has been used to develop a tool called the *Animation Choreographer*.
- *Polka* [16] is a software visualization toolkit that was designed to support end-user developed software visualizations. It allows to build algorithm and program animations. *Polka* is implemented in C++ on the top of the X windows System.
- *XTango* [14] is a general purpose algorithm animation system that supports programmers developing color, real-time, 2.5-dimensional, smooth animations of their own algorithms and programs. The focus of the system is on ease-of-use. *XTango* utilizes the path-transition animation paradigm which helps move animation design to an abstract, high level.
- ZADA [7] is a collection of interactive graphical animations of selected algorithms from the field of distributed algorithms and communication protocols

and resembles work going beyond isolated examples. The underlying animation system *Zeus* [3] has been extended from sequential algorithm visualization to cope with parallel algorithms.
- VADE [18] is a WWW-based environment for the visualization of distributed algorithm. The authors of VADE are also concerned with the causality problem. In their model, however, the price of the visualization consistency is an undesired feedback of the visualization onto the visualized distributed computation, which makes the system less suited for monitoring tasks.

6 Conclusion

In this paper, we have given an extended abstract of our work on the visualization of distributed algorithms [5] and we have empirically shown the feasibility of our concepts. Future work remains to improve our tool particularly to handle huge graphs as well as real-world networks. Parts of the tool are currently re-engineered with the goal of providing more intuitive interactions and displays as well as faster performance. We have used our tool to make several experiments supporting the analysis of several distributed algorithms. We think that our tool is useful as well for educational purposes (in order to explain the execution of distributed algorithms to students) as for researchers and practioneers in distributed systems (who require tool support for their tests and experiments).

Acknowledgments

Many thanks to *Yves Métivier* who has patiently answered several dozens of questions.

References

1. Hagit Attiya & J.Welch: *Distributed Computing.* McGraw-Hill, 1998
2. Marc Brown: *Algorithm Animation.* MIT Press, 1988
3. Marc Brown: *Zeus — A System for Algorithm Animation and Multi-View Editing.* VL'91 Proceedings in Visual Languages, pp.4-9, Oct.1991
4. GETGRATS: `http://www.di.unipi.it/~andrea/getgrats/`
5. Stefan Gruner & Mohamed Mosbah & Michel Bauderon: *A New Tool for the Simulation And Visualization of Distributed Algorithms.* LaBRI Report 1245-00, Université Bordeaux 1, October 2000. Available at http://dept-info.labri.u-bordeaux.fr/~mosbah/VSAD
6. Leslie Lamport: *Time, Clocks and the Ordering of Events in a Distributed System.* Communication of the ACM, Vol.21 No.7, pp.558-565, ACM Press 1978
7. A. Mester & H. Krumm: *Animation of Protocols and Distributed Systems.* To appear in: Journal of Computer Science Education, Special Issue on Educational Case Studies on Protocols and Distributed Systems, 2000
8. Yves Métivier & Igor Litovsky & Éric Sopena: *Graph Relabelling Systems and Distributed Algorithms.* In G.Rozenberg et al. (Eds.): Handbook of Graph Grammars and Computing by Graph Transformation,

Vol.3, Chpt.1, pp.1-56. World Scientific, Singapore 1999. Available at `http://www.worldscientific.com/books/compsci/4181.html`

9. Yves Métivier: *Graph Relabelling Systems — A Tool for Encoding, Proving and Studying Distributed Algorithms.* Proceedings of the French Workshop FAC'2000 (Formalisation des Activités Concurrentes), Toulouse 2000. See `http://www.irit.fr/FAC2000`
10. Yves Métivier & Nasser Saheb & Akka Zemmari: *Randomized Rendez-Vous.* LaBRI Report 1228-00, Université Bordeaux 1, January 2000. Available by `yves.metivier@labri.u-bordeaux.fr`
11. Yves Métivier & Pierre-A. Wacrenier: *A distributed Algorithm for Computing a Spanning Tree in Anonymous T-Prime Graphs.* LaBRI Report 1229-00, Université Bordeaux 1, January 2000. Available by `yves.metivier@labri.u-bordeaux.fr`
12. U. Montanari, M. Pistore & F. Rossi: *Modeling Concurrent Mobile and Coordinated Systems via Graph Transformation.* Vol.3, Chpt.4, pp.189-268 in H. Ehrig, H. Kreowski, U. Montanari & G. Rozenberg (Eds.): Handbook of Graph Grammars and Computing by Graph Transformation. World Scientific Publ., Singapore 1999
13. Blaine A.Price & Ronald M.Baeker & Ian S.Small: *A Principled Taxonomy of Software Visualization.* Journal of Visual Languages and Computing, Vol.4 No.3, pp.211-266, 1993
14. John Stasko: *Animating Algorithms with XTango.* SIGACT News, Vol.23 No.2, pp.67-71, 1992
15. John Stasko & Eileen Kraemer: *The Visualization of Parallel Systems — An Overview.* Journal of Parallel and Distributed Computing, Vol.18 No.2, pp.105-117, 1993
16. John Stasko & Eileen Kraemer: *A Methodology for Building Application-Specific Visualizations of Parallel Programs.* Journal of Parallel and Distributed Computing, Vol.18 No.2, pp.258-264, 1993
17. John Stasko: *The PARADE Environment for Visualizing Parallel Program Executions — A Progress Report.* Technical Report GA-GIT-GVU-95-03, The Graphics Visualization and Usability Center, Georgia Institute of Technology, Atlanta, 1995
18. Ayellet Tal & others: *Algorithm Visualization for Distributed Environments.* IEEE Symposium on Information Visualization '98, 1998. Available at `http://www.ee.technion.ac.il/~ayellet/Ps/dist.ps.gz`
19. Gerard Tel: *Introduction to Distributed Algorithms.* Cambridge University Press, 2000