# A New Tool for the Simulation And Visualization of Distributed Algorithms

Stefan Gruner | Mohamed Mosbah | Michel Bauderon
{stefan|mosbah|bauderon}@labri.u-bordeaux.fr

LaBRI, Université Bordeaux 1
351 Cours de la Liberation
**F** – 33405 **Talence** Cedex

**Abstract**

In this report [1] 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 by Yves Metivier. 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. An $\alpha$-version of the tool will be made available to the public soon.

[1]Supported by the TMR network GETGRATS. Contributed to the Dagstuhl-Seminar *Semi-Formal and Formal Specification Techniques for Software Systems* at Schloß Dagstuhl, Internationales Begegnungs- und Forschungszentrum für Informatik der Universität des Saarlandes (D), October $8^{th}$-$13^{th}$ 2000. [Dagstuhl-Seminar No.00411, hosted by H.Ehrig/G.Engels/F.Orejas/M.Wirsing as reported in Dagstuhl-Report No.288] See `http://www.dagstuhl.de/DATA/Seminars/00/`

# Contents

# 1 Introduction: Distributed Computing

In distributed computing [1][19], information is not processed by a single machine, but by several or many machines instead which share their work and keep in touch via communication links. In **Fig.1**, a (not complete) taxonomic tree of computing is sketched wherein the position of distributed computing in several possibilities is shown.

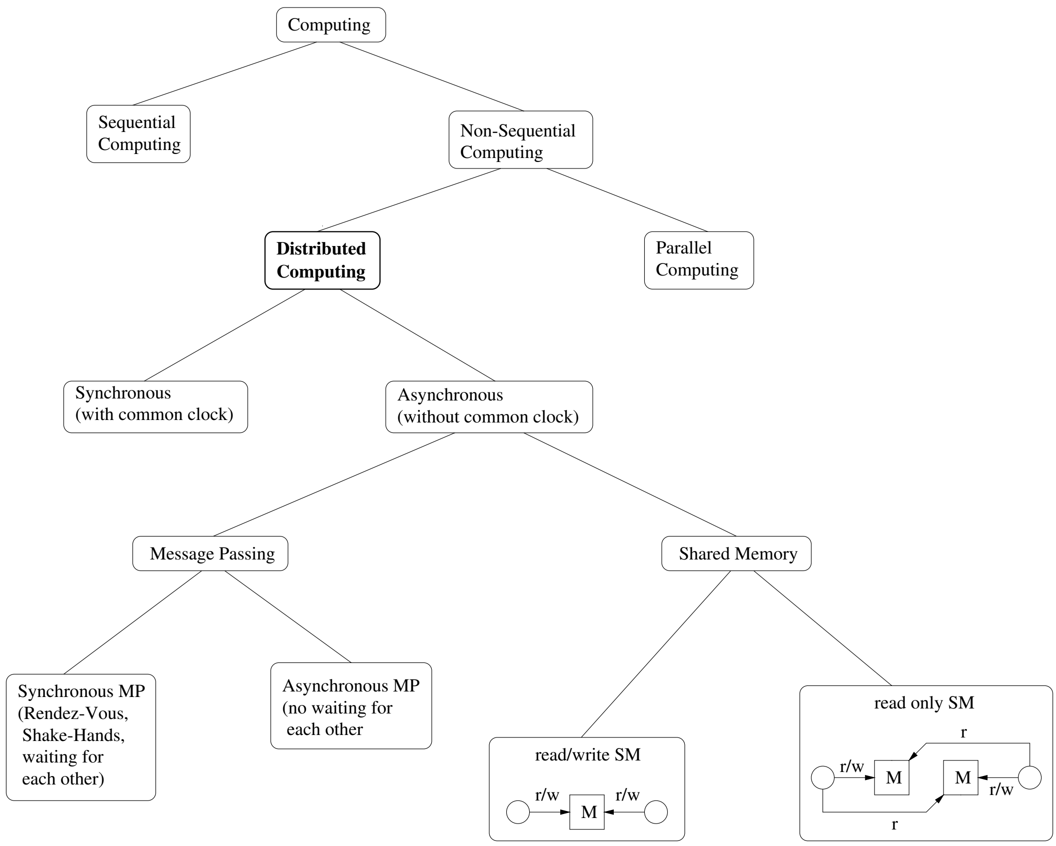


Figure 1: *a coarse taxonomy of computing*

The links of distributed computation to various fields of science as well as to our everyday life are manifold. In *Neuro-Science*, for example,

> "We have said good-bye to the old idea that our brains would operate as reactive machines with hierarchic structures of information processing. Meanwhile we have recognized that nowhere in our brains a place can be identified at which a coherent interpretation of our world would emerge, at which our decisions would be made, or at which our motoric actions would be planned. In opposite, the whole system is extremely distributed and

information is processed massively in parallel." [2] [13]

*Wolf Singer*,
Max-Planck-Institute for Neuro-Research

In the field of *Artificial Life*, one can observe *swarm intelligence* emerging from the cooperative behaviour of many primitive agents as for example artificial ants, artificial birds or artificial fish. In *Computer Science*, the *neural nets* and the *world wide web* are prominent motivations for studying the possibilities and limitations of distributed computation. Moreover, distributed systems have gained much *economical importance* due to an increasing number of web based applications and enterprise wide cooperating software. 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.

## 1.1 Technical Aspects and Obstacles

However, the development of distributed systems is yet not well understood. The design, validation, verification and debugging remain a hard task for most programmers and computer scientists. 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 assist the development of distributed applications. These tools include debuggers, performance monitoring, execution analysis, and other aids.

## 1.2 Contribution of our Work

Our work has focus on a *tool for visualizing and animating* distributed algorithms. 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 [8] 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.[3]

[2] Our translation from the German original

[3] Such principally unpredictable systems are called *computationally irreducible*.

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.

### 1.3 Context of our Contribution

An important property of an algorithm visualization system is its scope. There are special systems developed for the needs of a special project, 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 Metivier et al. [7][8][10].

### 1.4 Organization of this Report

In section 2 we introduce a basic model which is necessary to understand the principles of our approach. In section 3 the implementation of our tool is described. In section 4 we report on several experiments which we have performed in order to test the behaviour of the tool and of several distributed algorithms. After the "Look & Feel" of our tool is provided, we revisit our model in section 5 and describe some extensions of it according the actual implementation of our tool. In section 6 we treat the important problem of the visualization consistency, which means a proper relation between the images shown by the visualizer and the actual events on the net. Related work is discussed in section 7. Section 8 concludes the report and gives some hints to open problems and future work.

## 2 The Basic Model

In this section, our notion of *distributed computation* is explained with respect to other notions of distributed or parallel computation. This also implies a slightly different notion of *algorithm* compared to its classical notion. We describe the structures in which distributed computation takes place as well as the entities of which those structures consist. As we assume an *asynchronous mode* of distributed computation in our model, we explain the notion of asynchronicity in contrast to the notion of synchronicity.

## 2.1 Distributed Computation

In the classical way of distributed or *parallel* computation, we have a program that shall be executed and several processors which shall execute this program. One of these processors is a *master* which organizes the distribution of the parallelizable program fragments on the processors and collects their sub-results in order to produce the final result. In **Fig.2**, this situation is sketched.

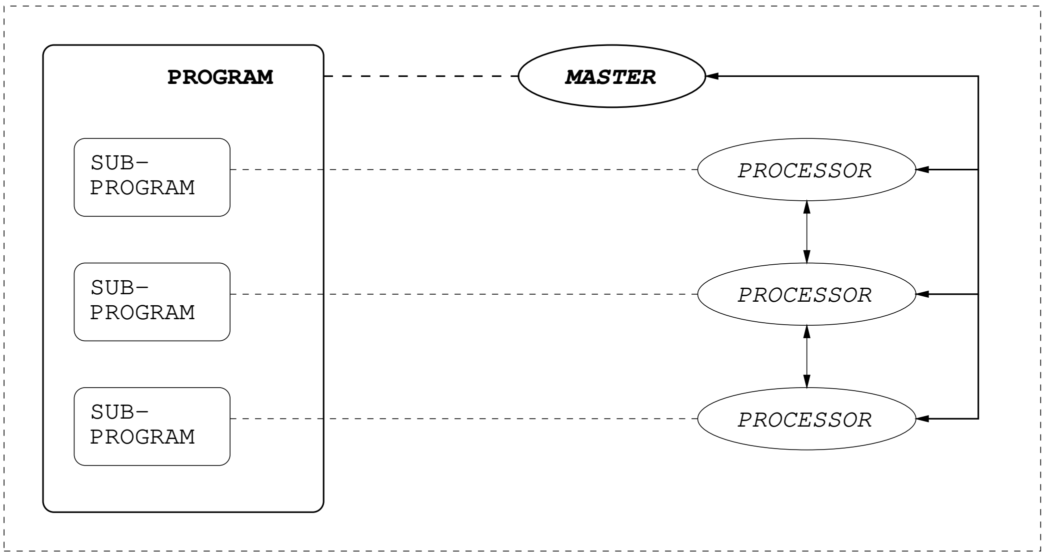


Figure 2: *classical distributed computing organized by a master*

The situation is different in *autonomous* distributed computation which our concept is based on. Here, 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. In **Fig.3**, this situation is sketched.

**Remarks:** A multi-processor computer is an example of master-driven distributed computation. In this case, the operating system of the machine plays the role of the master.

In a master-driven system, the role of the master may be exchanged among the involved processors or *agents*. This is, for example, the case in client-server-based, view-point-oriented multi-user software development environments [4].

*Neural networks* [12][20] and *cellular automata* [21] are well-known examples 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

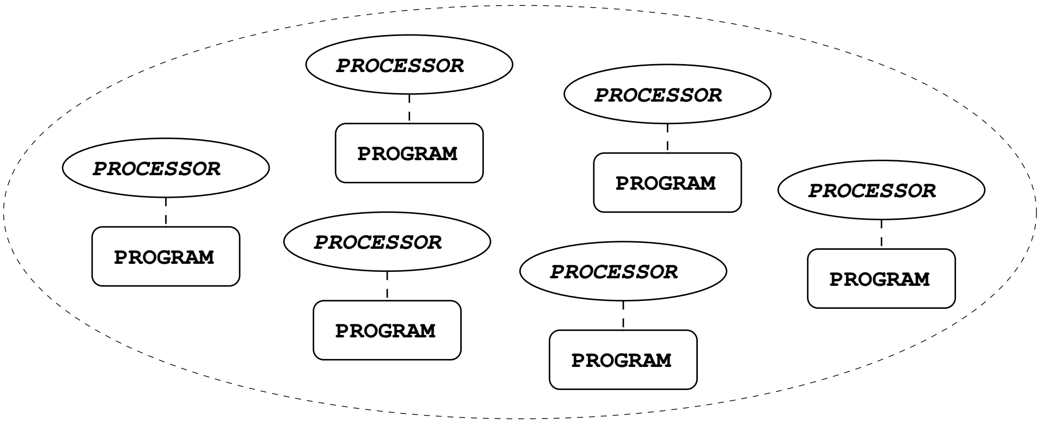


Figure 3: *autonomous distributed computing organized by itself*

each occurring algorithm executed by its corresponding processor. There is no meta-algorithm explicitly describing the system behaviour emerging from the co-operation of the autonomous processors or entities.

Moreover, 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* with nodes and edges as depicted in **Fig.4**.

**Nodes:** We assume that each node in the graph represents an entity (or: an agent) which is able to compute and which is able to send and receive *messages*. 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.5**. Of course, a node's current state $s \in S_k$ is not necessarily

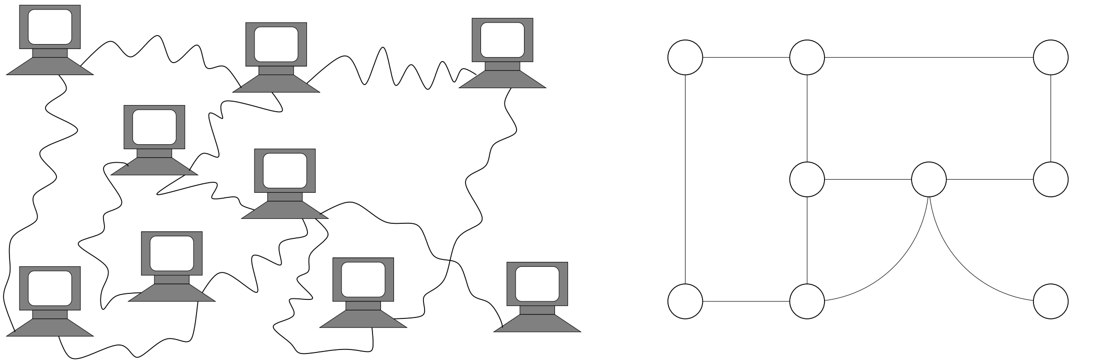

Figure 4: *net and graph representation*

invariant: usually it will change from time to time during an ongoing distributed computation.

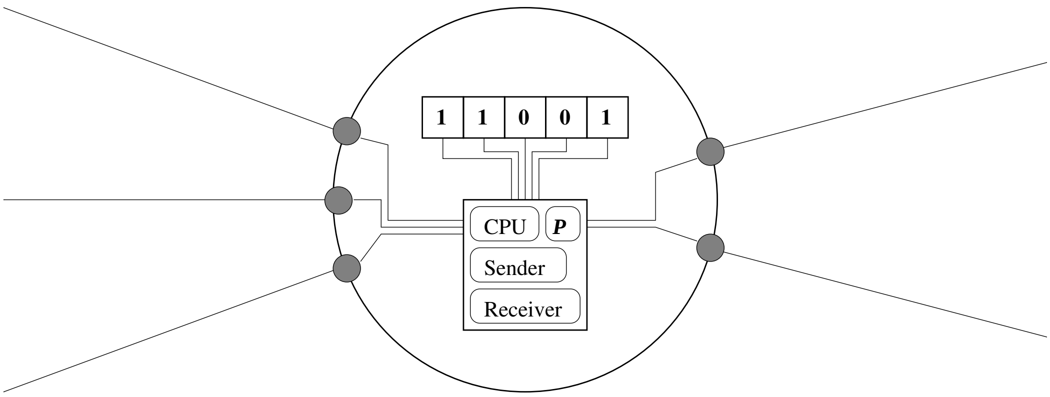


Figure 5: *node (processor) with program $P$, current state $s = 11001$, and $C = 5$*

**Edges:** We assume that each edge of the graph represents a bi-directional message *channel*. We assume further that messages travelling through the channel cannot overtake each other which means that the channel operates in a *FIFO* mode. For our simple model we assume moreover that the channel is *perfect* such that no message can be lost or changed while travelling through the channel. Finally we assume *full duplex* communication which means that two messages can travel in both directions at the same time without disturbing each other. (By assuming full duplex communication it is not necessary to model the channel as an active entity with a semaphore blocking technique.) The situation is sketched in **Fig.6**.

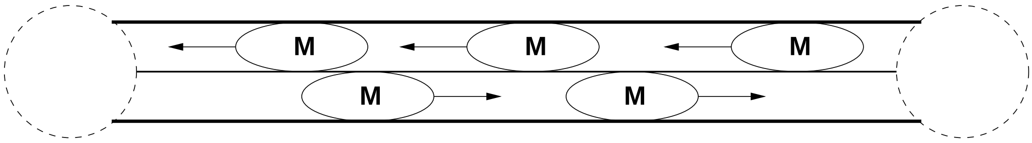


Figure 6: *full duplex FIFO channel with travelling messages*

**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 $e$. (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 $e$.)

**Remark on Self-Consciousness:** When a network of distributed computation is represented as a graph, one can ask the question if such a system is able to recognize some of its own graph-theoretic properties, for example: "Am I Hamiltonian?", "am I cycle-free?", "am I bi-partite?", and the like. The question if a graph can recognize its own properties by distributed autonomous computation is, for example, treated in [8].

**Section Summary:** In this section we have regarded a simple model of autonomous distributed computing in networks which are described by graphs. The nodes of the graphs are represent computing entities while the edges represent perfect full-duplex FIFO communication channels between them. The communication in such a network is assumed to be asynchronous.

# 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 build a network easily, and to visualize the execution of a distributed algorithm. The visualization provides colors, animation and interaction with the user. The tool is written with JAVA which makes it platform independent. 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.7**. These modules are well-separated according to the principles of software construction such that the modification of one component involves only small changes for the rest of the system.

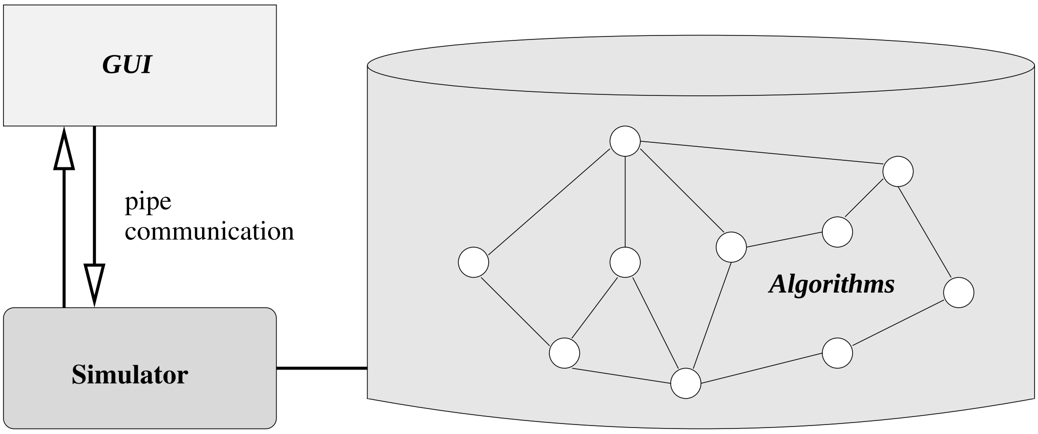


Figure 7: *architecture of the prototype*

## 3.1 The Graphical User Interface

The editor allows the user to construct a network by drags and drops as shown in **Fig.8**. Most common commands are available. The user can add, delete, or select vertices, edges or subgraphs. Moreover, the visual attributes of a vertex —labels, colors, and shapes— can also be set by the user. Most buttons are intuitive.


Figure 8: *user interface of the prototype*

Once the network constructed, 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. Moreover, the status of edges and nodes are updated on-line. The user can interact with the visualization of the algorithm by stopping or suspending the execution. 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.

The sequence of **Fig.9**, **Fig.10**, **Fig.11**, and **Fig.12** shows an example of the distributed algorithm that generates a random spanning tree as described in [10]. Detailed descriptions of several experiments are given in Section 4.

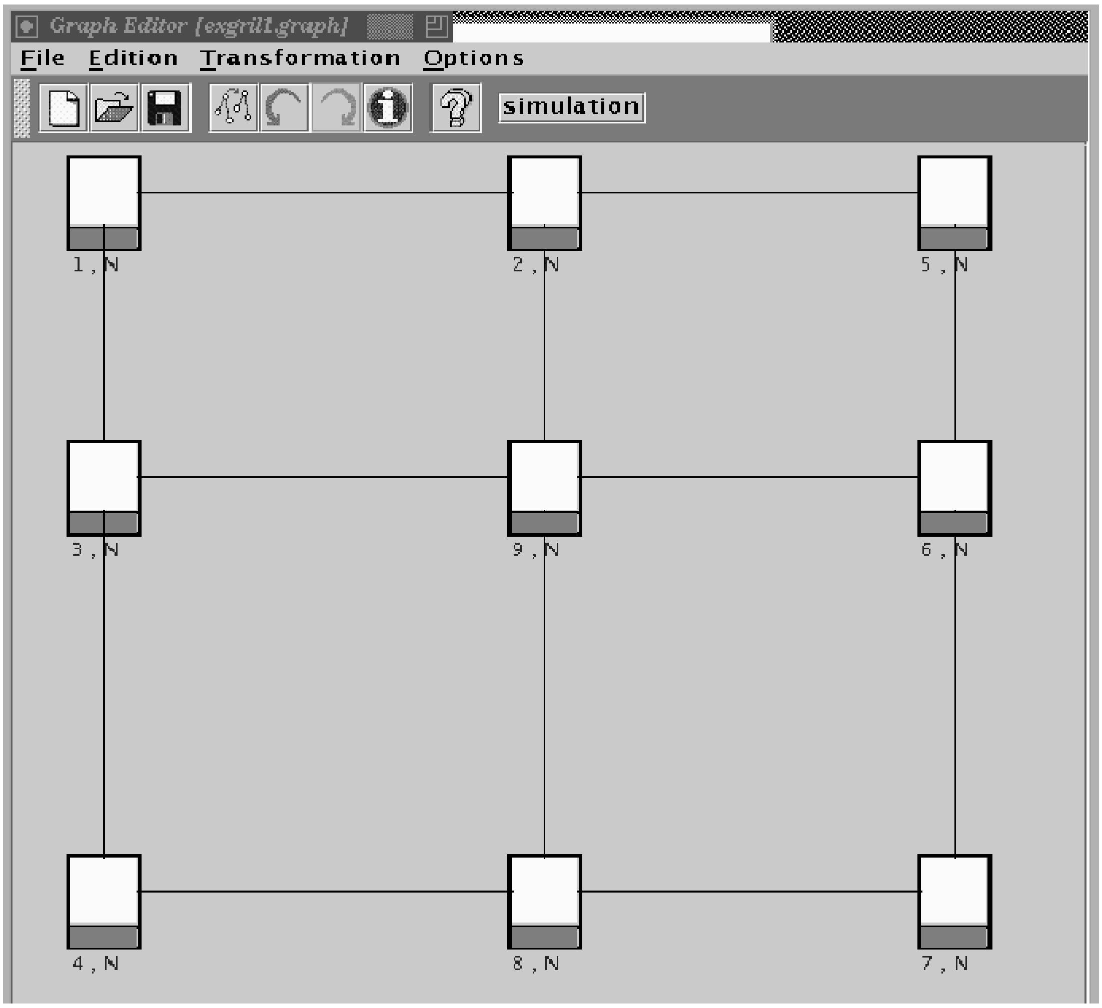


Figure 9: *initial step: before the algorithm has started (0)*

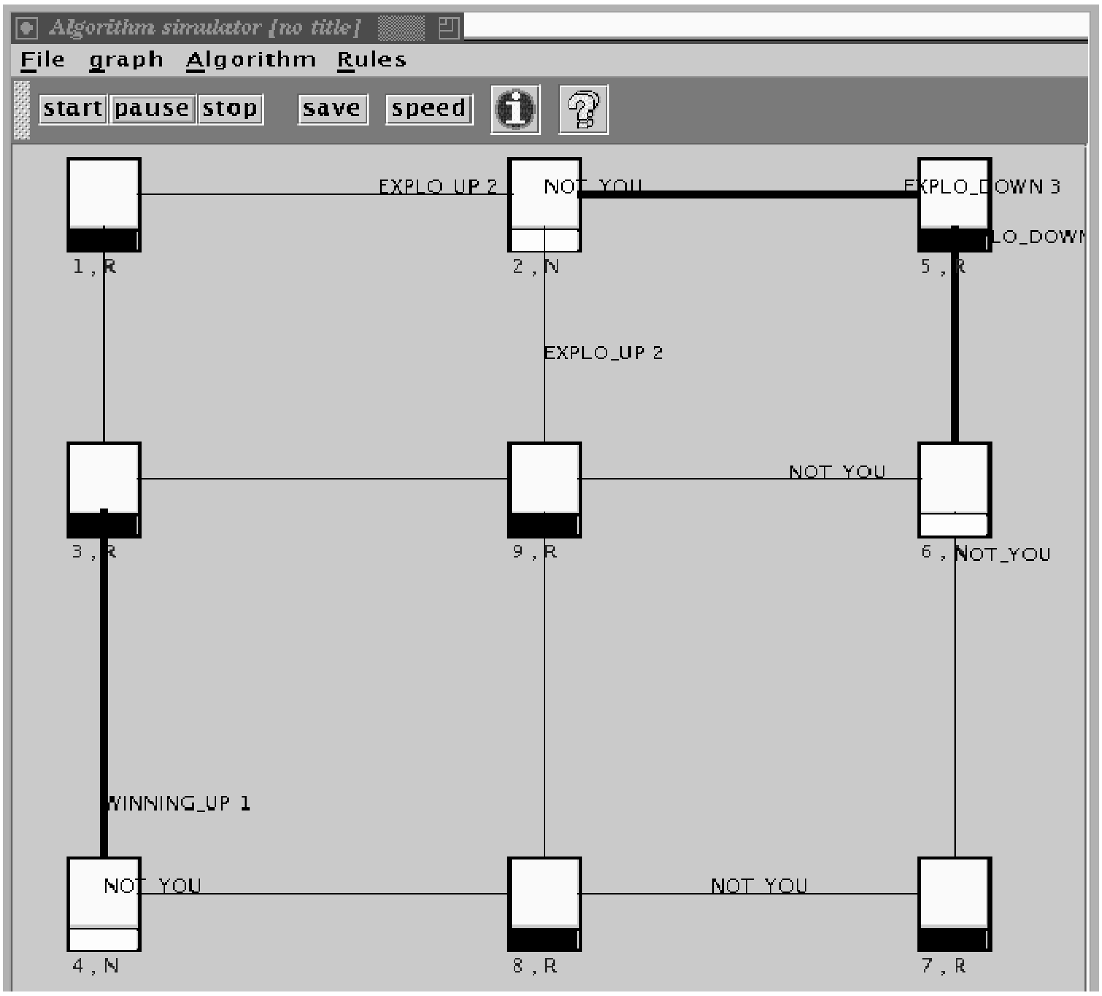


Figure 10: *visualization of the algorithm execution (1)*

## 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:

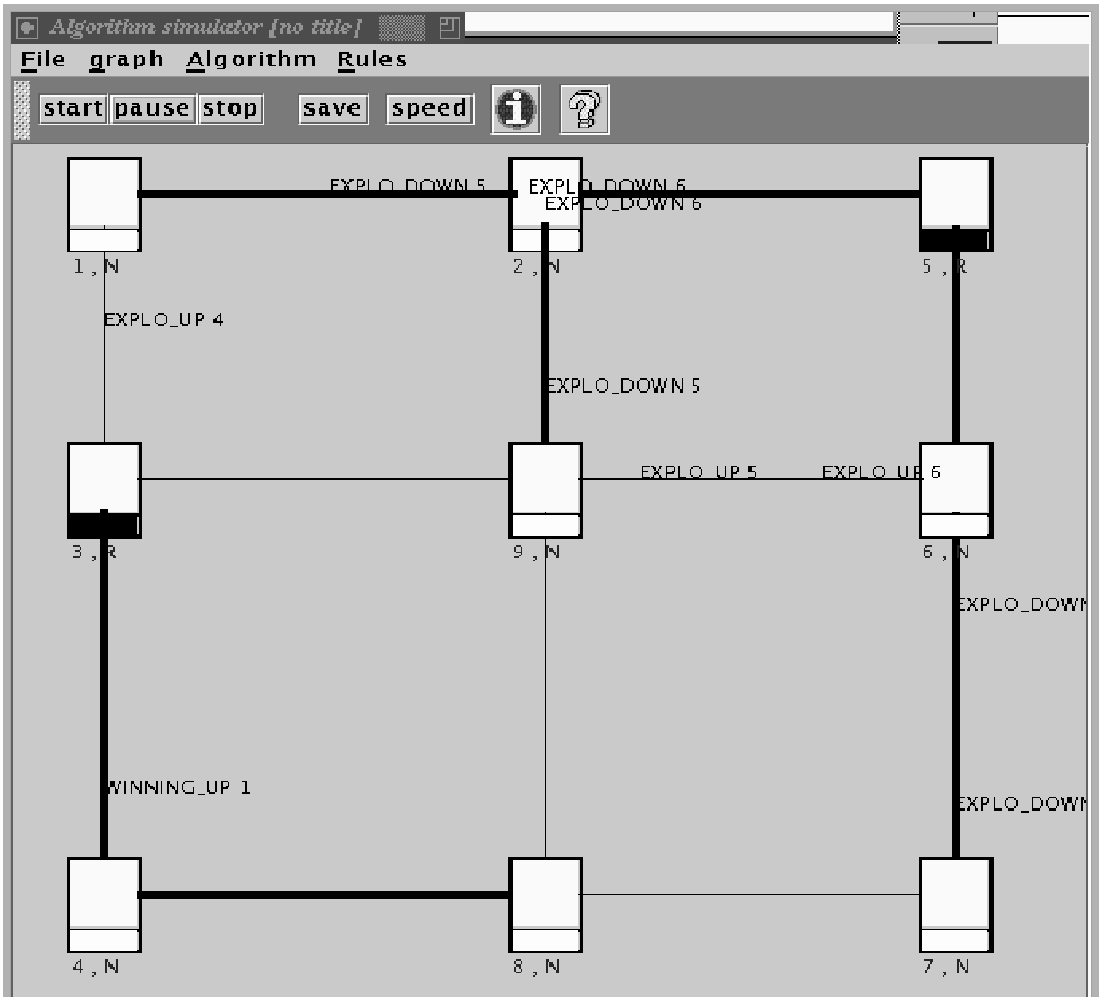


Figure 11: *visualization of the algorithm execution (2)*

- **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

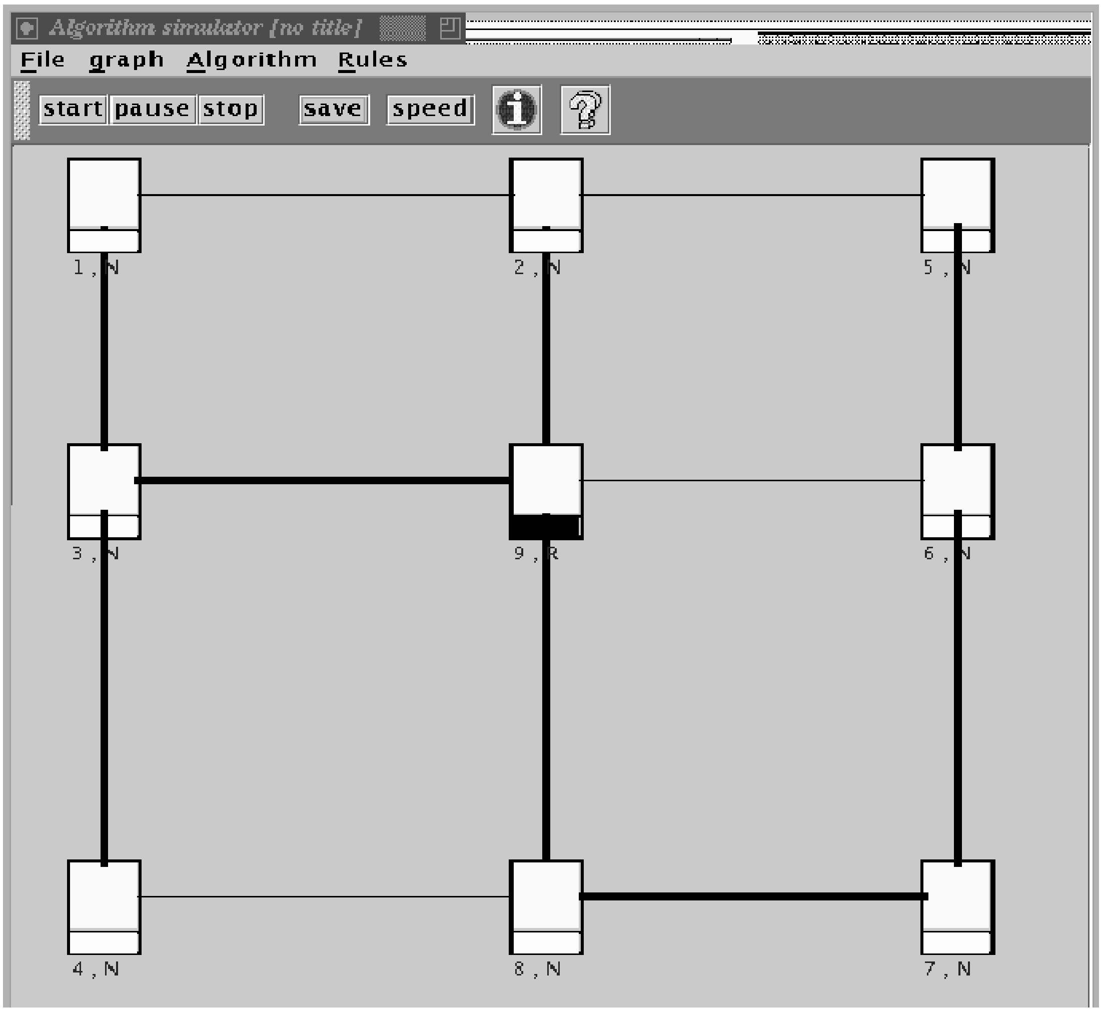


Figure 12: *result of the computation (3)*

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 [9].

Many standard distributed algorithms are already implemented. These include the *leader election* and *star synchronization* [8] as well as *spanning tree computation* [10]. 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 [7]. 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.

## 4.1 Motivation

In his report on a distributed algorithm for computing *spanning trees* in anonymous T-prime graphs, Metivier has stated that

> "It remains to study for a given non-T-prime graph the probability that a deadlock happens. We conjecture that this probability tends to 0 when the size of the graph increases to the infinity." [10]

The experiments reported in this section may be regarded as first steps into such a study. But also other algorithms, for example the *leader election* can serve as "high energy fuel" for driving the engine of our prototype to its very limits in our behavioural tool experiments.

## 4.2 Preliminaries, Results and Interpretations

Before performing experiments with the tool one has to recall 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".

### 4.2.1 The Quality of Random Generation

Our initial experiment is dedicated to the question of how good is the random behaviour of our prototype. This is a very important question, because many algorithms executed in distributed computation are based on random numbers.

**Preliminaries:** As the basis of the experiment, we take a fully connected triangle Graph $\triangle$ with three nodes $A, B, C$. Election, ElectionNet and SpanningTree are three different distributed algorithms which eventually return one certain node as output. Thus: because the triangle graph $\triangle$ is perfectly symmetric, we should expect that all nodes $A, B, C$ are returned equally often as output by each algorithm in a random environment, supposed that the experiment is repeated often enough.

**Result:** Here are the the results (how often output) of $60 + 60 + 60 = 180$ repetitions, with 60 repetitions for each of the three algorithms:

$A$: Election 19, ElectionNet 29, SpanningTree 26, $\Sigma$ 74

$B$: Election 24, ElectionNet 14, SpanningTree 13, $\Sigma$ 51

$C$: Election 17, ElectionNet 17, SpanningTree 21, $\Sigma$ 55

**Interpretation:** Obviously, the theoretical medium-value is $\hat{m}_{x,y} = 20$ per node and algorithm, and $\hat{m}_x = 60$ per node for the whole experiment, whereby $x \in \{A, B, C\}$ and $y \in \{\text{Election}, \text{ElectionNet}, \text{SpanningTree}\}$.

To measure the quality of this result, we take the theoretical and empirical *variances* into account. Based on 60 repetitions (for each algorithm $y$), the theoretical variance (for each node $x$) has the value $\sigma^2_{x,y} \approx 13$. In the same way we can evaluate $\sigma^2_x \approx 40$ (for each node $x$) based on 180 repetitions for the whole experiment.

Please note that the numbers of outputs in the experiments are *not stochastically independent*, as $A_y + B_y + C_y = 60$ for each $y$ and $A + B + C = 180$ for the whole experiment. For this reason, we *must not compute the common* empirical variances $s^2$ of $A, B, C$ together for comparison! However, it seems legal to compute the occurring minimal and the maximal particular empirical variances $s^2_{x,y}$ and to compare those values with the theoretical variance. Doing so, we obtain:

- $y =$ Election:
  $s^2_A = min = (19 - 20)^2 = 1 < 13,\ \ s^2_B = max = (24 - 20)^2 = 16 > 13.$
- $y =$ ElectionNet:
  $s^2_C = min = (17 - 20)^2 = 9 < 13,\ \ s^2_A = max = (29 - 20)^2 = 81 > 13.$
- $y =$ SpanningTree:
  $s^2_C = min = (21 - 20)^2 = 1 < 13,\ \ s^2_B = max = (13 - 20)^2 = 49 > 13.$
- all algorithms together:
  $s^2_C = min = (55 - 60)^2 = 25 < 40,\ \ s^2_A = max = (74 - 60)^2 = 196 >> 40.$

One may perhaps interpret these results as "not too bad", even though one certainly has to admit that the random behaviour could be improved in a future version of our tool.

### 4.2.2 Spanning Tree Computation in a String Graph

A string graph $G = \square\!-\!\square\!-\!\square\!-\!\square$ consisting of 4 serially connected nodes $A, B, C, D$ is a simple example of a *non*-T-prime graph. For this reason Metivier's SpanningTree algorithm may sometimes fail when applied to this graph, because SpanningTree is explicitly designed to operate on T-prime graphs.[4] In this experiment, we have made 200 repetitions of SpanningTree on $G$ and observed the distribution of results, especially with respect to failure or success.

**Preliminaries:** Based on a combinatorial distinction between $A, B, C$ and $D$, one can distinguish eight possible results of an application of SpanningTree to $G$ which are sketched in **Fig.13**: outputs of types (a), (b), (c) and (d) are successful application results, whereas outputs of types (e), (f), (g) and (h) have to be regarded as failure. With 200 repetitions and eight possible outputs we would have 25 counts per possibility if the result were perfectly equi-distributed.

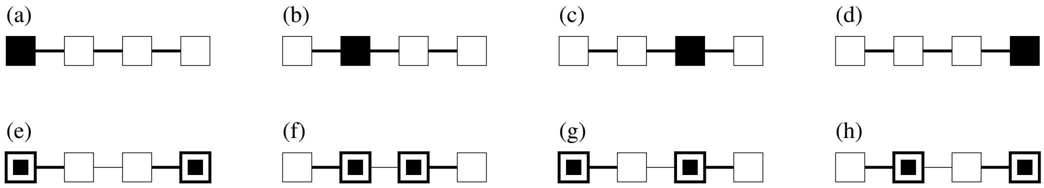


Figure 13: *possible outputs of SpanningTree on G*

Please note that the local behaviour of SpanningTree depends on the connectivity of the involved nodes (i.e. the amount of attached edges) such that we may *not* assume an equi-distribution of the eight results a-priori; this is because of nodes $A, D$ having not the same connectivity as $B$ and $C$. We might assume a-priori, however, that results which are structurally isomorphic with each other —(a) with (d), (b) with (c), and (g) with (h)— occur with the same frequency in the ideal case. Thus, combinatorial success-failure-ratio is $(4 : 4) = (1 : 1)$. Because the string has size 4, a failure output consists of 2 part-trees each of size 2 as shown in Fig.13.

**Result:** As a result of our experiment we observed the following output frequencies: (a)24, (b)45, (c)50, (d)16, (e)12, (f)15, (g)16 and (h)20. However in two cases we did not obtain any output at all as the tool faded with some runtime error, thus: $(\perp)2$.

**Interpretation:** As visualized in **Fig.14**, we have a success rate of $135/200 \approx 67\%$ which seems not bad for such a small graph. However, the quality-of-random result of the previous experiment must always be kept in mind! Grouping the results by structural isomorphism, Fig.14 shows also that the frequencies of structurally equivalent outputs seem to differ not too much from each other. Finally

[4]T for "Tree". For *non*-T-prime graphs, Metivier's spanning tree algorithm is both "LasVegas" *and* "MonteCarlo" in the sense that it might not necessarily terminate and in case of termination the result might not necessarily be good.

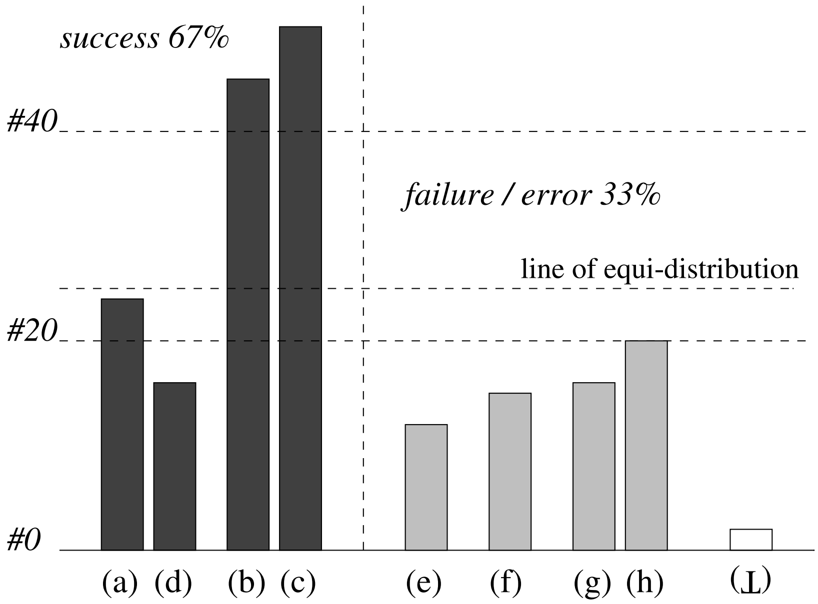


Figure 14: *distribution of outputs in the string graph experiment*

it seems not surprising that the output of the results (b) and (c) is considerably higher than of (a) and (d), which have in fact only a small theoretical probability due to the design properties of SpanningTree, which can be found in [10]. The empirical success-failure-quota of $(135 : 65) \approx (2 : 1)$ is considerably better than the combinatorial quota of $(1 : 1)$.

**Remarks on the Runtime Errors:** Due to the current architecture of the prototype (see section 3), there is a communication bottleneck between the rather fast computation layer and the rather slow GUI layer of our tool. This might lead to crashes especially when the speed parameter has been selected too high.

Moreover, one has to keep in mind that the JAVA pseudo-threads, which are used in the implementation of our tool, survive as half-live zombies after they have done their job; a large amount of such zombies in the system might also have bad influence on its behaviour.

Of course, also any other undetected software bugs in the prototype code might be the reason for misbehaviour.

### 4.2.3 Spanning Tree Computation in a Ring Graph

In order to get a better "feeling" for the results of the previous experiment, we bring it in the context of a similar experiment which is described in the following.

**Preliminaries:** Instead of a string graph we take a ring graph such that — as in the experiment with the triangle graph $\triangle$, but in contrast to the previous experiment— the connectivity is the same for each involved node. Instead of four nodes we take six: $A, B, C, D, E, F$, such that the number of possible outputs of SpanningTree is considerably bigger than in the previous experiment. In fact, we now have 36 success possibilities wherein a spanning tree is constructed. In total we have 79 output possibilities such that 43 of them are failures. Thus the combinatorial success-failure-ratio is now $(36 : 43) \approx (9 : 11)$ which is slightly worse than the ratio of $(1 : 1)$ in the previous experiment. 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.15** shows our ring graph with one example of a successful output and two examples of failures. We made $79 \times 3 = 237$ repetitions such that each of the 79 possible output would occur 3 times in an ideally equi-distributed result.

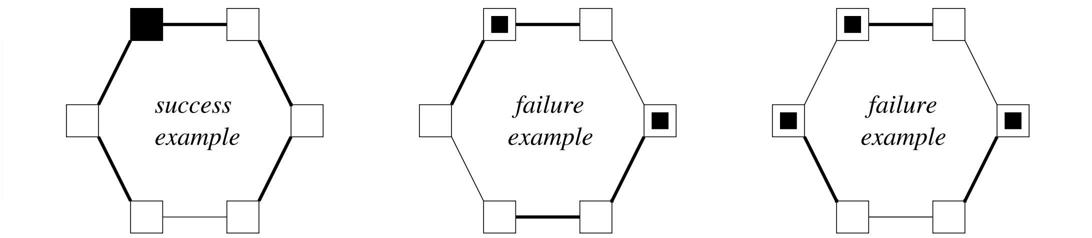


Figure 15: *ring graph with example outputs of success and failure*

**Result:** As depicted in **Fig.16**, only 43 of all the 79 output possibilities have been observed at all. In this experiment, no runtime error occurred. Success has been noticed in 172 cases, while failure was counted 65 times.

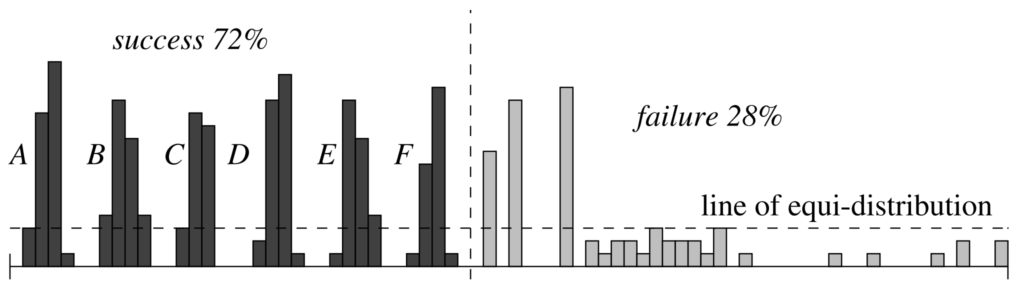


Figure 16: *distribution of outputs in the ring graph experiment*

**Interpretation:** The result is far away from equi-distribution because the probability, for example, that the head-node of a spanning tree is at the very edge of that tree is only very small. In fact, it is much more likely from the design properties of SpanningTree [10] that the head-node is located in the middle of the tree. We see this property reflected by the result very well.

The six dark grey "towers" in the success part of Fig.16 show that the nodes $A, B, C, D, E, F$ 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 the ring graph which equal connectivity for each of its nodes. This result might be regarded as another hint to the acceptable random behaviour of our prototype.

Failure outputs with 3 part-trees each of size 2 are very seldom; each of those occurrences is below the flat line of equi-distribution. However, failure outputs with 2 part-trees each of size 3 are more likely and seem to occur more often in fact. (Please have a look at Fig.15 again.)

Though the combinatorial success-failure-quota of about $(9 : 11)$ is worse than in the previous experiment, the empirically observed success-failure-quota of $(172 : 65) \approx (3 : 1) \approx 72\%$ is considerably better now. This might be regarded as a hint to Metivier's conjecture which is quoted above at the beginning of the section.

#### 4.2.4 Leader Election in Anonymous Ring Graphs

Having investigated the behaviour of the SpanningTree algorithm in the previous experiments, we now turn to the ElectionNet algorithm which elects a node in an anonymous context as further described in [8]. Our question is now how this random-based algorithm behaves as the size of the input graphs grows, thus we are asking for the *empirical complexity* of ElectionNet. The results of the empirical complexity investigation may provide motivation for later investigations concerning the theoretical complexity of this algorithm.

**Preliminaries:** We take ring graphs as in the previous experiment. We let the distributed algorithm operate until termination [5] and then we *count the number of messages $m$* exchanged during the distributed computation. We expect that the number of exchanged messages is not independent from the number of edges of the graphs, as messages are transmitted via the edges. The set of ring graphs constitutes the (al)most simplest graph class with respect to connectivity and node-edge-quota. In fact, the set of edges $E$ in ring graphs grows in the same order as the set of nodes $N$, thus: $\mathcal{O}(|E|) = \mathcal{O}(|N|)$, or, to be even more precise, $|E| = |N|$. We take rings of sizes $n = 3, \ldots, 9$ and we we make 15 repetitions per ring.

[5] In distributed computation in anonymous graphs, there is a slight difference between *implicit* termination and *explicit* termination [8] which, however, is not important for the performance of our experiment.

**Result:** The following numbers of exchanged messages $m$ have been counted during the repetitions of the experiment.

- $n = 3$: $19, 22, 22, 19, 18,\ 20, 18, 20, 18, 20,\ 20, 18, 18, 18, 20.$
- $n = 4$: $25, 35, 37, 34, 35,\ 34, 34, 33, 32, 33,\ 35, 34, 34, 34, 33.$
- $n = 5$: $51, 52, 57, 50, 57,\ 51, 51, 58, 55, 51,\ 50, 52, 50, 52, 52.$
- $n = 6$: $79, 82, 78, 73, 75,\ 79, 79, 73, 75, 79,\ 72, 74, 72, 73, 77.$
- $n = 7$: $110, 106, 102, 99, 100,\ 101, 99, 105, 107, 104,\ 102, 104, 104, 101, 102.$
- $n = 8$: $134, 138, 142, 134, 130,\ 135, 129, 129, 134, 130,\ 131, 135, 128, 129, 135.$
- $n = 9$: $164, 173, 165, 176, 168,\ 164, 163, 163, 169, 166,\ 169, 176, 170, 171, 164.$

**Interpretation:** For ring sizes $n = 3, \ldots, 9$ we can compute the mean-values $\hat{m}_n$ of counts, and we observe that no counter is far away from its corresponding mean-value. The results are considerably "dense", and these small variances seem to reflect the restricted communication possibilities in the poorly-structured rings with extremely low connectivity. Moreover we observe that the fraction $\frac{\hat{m}_n}{n^2}$ is almost constant ($\approx 2.1$) in the examined domain ($n = 3, \ldots, 9$):

- $n = 3,\ \hat{m}_3 \approx 19.333,\ \frac{\hat{m}_3}{3^2} \approx 2.148$
- $n = 4,\ \hat{m}_4 \approx 33.466,\ \frac{\hat{m}_4}{4^2} \approx 2.091$
- $n = 5,\ \hat{m}_5 \approx 52.600,\ \frac{\hat{m}_5}{5^2} \approx 2.104$
- $n = 6,\ \hat{m}_6 \approx 76.000,\ \frac{\hat{m}_6}{6^2} \approx 2.111$
- $n = 7,\ \hat{m}_7 \approx 103.066,\ \frac{\hat{m}_7}{7^2} \approx 2.103$
- $n = 8,\ \hat{m}_8 \approx 132.866,\ \frac{\hat{m}_8}{8^2} \approx 2.076$
- $n = 9,\ \hat{m}_9 \approx 168.066,\ \frac{\hat{m}_9}{9^2} \approx 2.074$

Thus, altogether this experiment might be regarded as a hint for a possible $\mathcal{O}(n^2)$ complexity of the random-based ElectionNet algorithm *on ring graph inputs.*

### 4.2.5 Leader Election in Anonymous Cliques

In a clique, also known as *complete graph*, every node is connected to every other node by exactly one edge. Here, the number of edges $E$ grows much faster than the number of nodes, namely $\mathcal{O}(|E|) = \mathcal{O}(|N|^2)$, or, to be even more precise: $|E| = \frac{k(k+1)}{2}$ where $k = |N| - 1$ (which is according to the well-known formula for the chain-sum $i = 1 + \cdots + k$). Thus, the set of all cliques belongs to the graph classes with highest connectivity.

Having studied the behaviour of ElectionNet for poorly-connected rings in the previous experiment, we now test the behaviour of the same algorithm on some highly-connected cliques. With this comparison, we hope to find some upper bounds and lower bounds of the average complexity of ElectionNet on "usual" graphs which seem to have higher connectivity than rings but lower connectivity than cliques in many practical cases.

**Preliminaries:** As done in the previous experiment, we count the exchanged messages along the edges of the graph until termination in 15 repetitions per clique. As in the previous experiment, the clique size grows from $n = 3$ to $n = 9$ in terms of nodes, which means from $e(n) = 3$ to $e(n) = 36$ in terms of edges according to the formula of above. For $n = 3$, a clique is also a ring such that we can refer to the previous experiment in this special case.

**Result:** The following numbers of exchanged messages $m$ have been counted during the repetitions of the experiment.

- $n = 3$: see previous experiment (clique-3 = ring-3).
- $n = 4$: $102, 113, 108, 114, 108, 114, 111, 112, 110, 112, 108, 109, 111, 110, 109$.
- $n = 5$: $348, 234, 302, 260, 280, 274, 278, 275, 302, 317, 372, 288, 373, 373, 287$.
- $n = 6$: $477, 460, 457, 637, 555, 554, 522, 495, 639, 568, 522, 589, 530, 641, 679$.
- $n = 7$:
  $1031, 838, 1919, 1280, 1018, 1087, 987, 1152, 1127, 892, 1487, 688, 882, 1179, 874$.
- $n = 8$:
  $1203, 543, 2077, 2882, 1210, 879, 1609, 1525, 1399, 2204, 871, 2361, 1742, 2321, 778$.
- $n = 9$:
  $3490, 4316, 4064, 2885, 1536, 4895, 2271, 3455, 2342, 2312, 973, 2929, 1394, 2579, 2172$.

**Interpretation:** In contrast to the previous experiment, the number of exchanged messages grows considerably faster than $\mathcal{O}(n^2)$, because the communication depends stronger on the number of edges than on the number of nodes.

Also in contrast to the previous experiment, the variance of counting is widespread with respect to the according mean-values here, which makes it rather difficult to predict a reliable complexity formula — nevertheless we try. The observed wide range of message counts might reflect the vast amount of different communication possibilities in a clique (which, by the way, comprises also the possibility of efficient ring-communication with rather few messages until termination).

Similar to the previous experiment, however, we can observe that that the fraction $\frac{\hat{m}_n}{(e(n))^2}$ is in close neighbourhood to $\approx 2.4$ for the investigated domain of $n = 3, \ldots, 9$. This value is quite similar to the according value of the previous experiment (and might be, therefore, perhaps regarded as a characteristic value of the ElectionNet algorithm?). Here however, the variance of this value is bigger than in the previous experiment which might be due to the higher variances of the counting results in this case.

- $n = 3,\ e(3) = 3,\ \hat{m}_3 \approx 19.333,\ \frac{\hat{m}_3}{3^2} \approx 2.148$
- $n = 4,\ e(4) = 6,\ \hat{m}_4 \approx 104.066,\ \frac{\hat{m}_4}{6^2} \approx 2.890$
- $n = 5,\ e(5) = 10,\ \hat{m}_5 \approx 304.200,\ \frac{\hat{m}_5}{10^2} \approx 3.042$
- $n = 6,\ e(6) = 15,\ \hat{m}_6 \approx 555.000,\ \frac{\hat{m}_6}{15^2} \approx 2.446$
- $n = 7,\ e(7) = 21,\ \hat{m}_7 \approx 1096.066,\ \frac{\hat{m}_7}{21^2} \approx 2.485$
- $n = 8,\ e(8) = 28,\ \hat{m}_8 \approx 1573.600,\ \frac{\hat{m}_8}{28^2} \approx 2.006$
- $n = 9,\ e(9) = 36,\ \hat{m}_9 \approx 2774.333,\ \frac{\hat{m}_9}{39^2} \approx 2.141$

Thus, this experiment might be regarded as a hint for a possible $\mathcal{O}(|E|^2) = \mathcal{O}(n^4)$ complexity of the random-based ElectionNet algorithm *on clique inputs*.

As an overall result of this experiment together with the previous experiment one might perhaps expect an $\mathcal{O}(n^3)$ complexity of the ElectionNet algorithm when applied to "average" graphs being neither cliques nor rings ($n^4 > n^3 > n^2$). This interpretation is depicted in **Fig.17**.

### 4.2.6 Ring Size and Success Rate with Spanning Tree

We now turn back to Metivier's question what happens with SpanningTree and *non*-T-prime graphs when the size of the input graph is growing to the infinity. Unfortunately our small tool prototype is yet not powerful enough to deal with larger graphs, nevertheless we can tackle this question at least in a restricted domain.

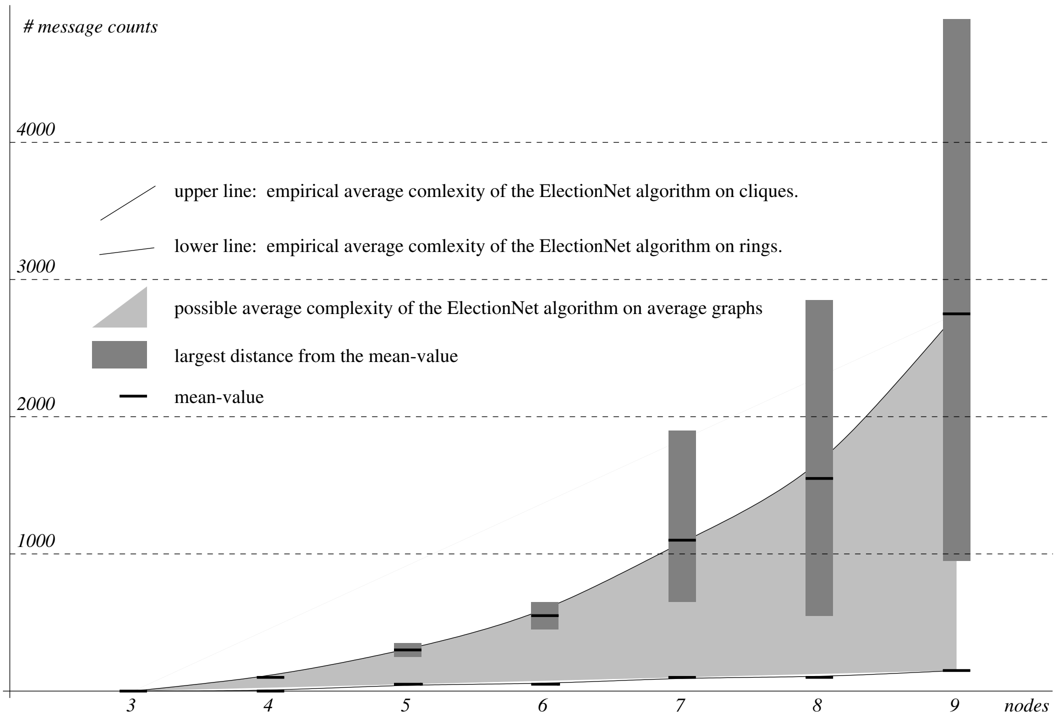


Figure 17: *results and interpretation of the ElectionNet algorithm*

**Preliminaries:** As non-T-prime graphs, we take (again) ring graphs of sizes $4, 6, 8, 10, 12$ (in number of nodes), and for each ring we repeat the SpanningTree algorithm 100 times. In contrast to the earlier experiments, we are now not interested in the precise gestalt of each output: we only count the numbers of success and failure. For convenience, we take the result of our earlier ring graph experiment for the case of ring of size 6.

**Result:**

- Ring-4: 82 success, 16 failure, 2 runtime errors.
- Ring-6: success rate $\approx 72\%$, see earlier experiment.
- Ring-8: 86 success, 14 failure.
- Ring-10: 77 success, 33 failure.
- Ring-12: 84 success, 16 failure.

**Interpretation:** Obviously we can neither verify nor falsify Metivier's forecast with such a simple —yet tedious!— experiment in general. But we might state that *if* Metivier is right, there is at least one class of graphs, namely the class of small rings, for which the success rate of SpanningTree does not grow very fast with the size of the graphs. Here, a success rate of about 80% seems to be the limit that one has to accept. However, we should not worry about this result, because the probability that SpanningTree fails *twice* on the same input is only $(0.2)^2 = 4\%$ under these circumstances, and a double application of SpanningTree seems not to be too expensive in this case.

#### 4.2.7 Grid Size and Success Rate with Spanning Tree

With this experiment, we want to provide a context of comparison for the previous one. Having a ring of size $2n$ ($n \in \mathbb{N}$) the nodes of which are sequentially numbered in $0, \ldots, 2n-1$, then we can construct a $n \times 2$-grid from this ring by adding edges between node $(0+1) mod\ 2n$ and node $(0-2) mod\ 2n$, between node $(0+2) mod\ 2n$ and node $(0-3) mod\ 2n$, and so forth until there is no edge more to add.

**Preliminaries:** Analogous to the previous experiment, we perform 100 repetitions of SpanningTree each on grids of sizes $2\times2$, $3\times2$, $4\times2$, $5\times2$, $6\times2$, (which are non-T-prime graphs as well) and we count success and failure again. As the $2\times2$-grid is the same object as a 4-ring, we are lazy enough to take the result of the previous experiment in this special case.

**Result:**

- Grid $2 \times 2$: success rate 82%,
  see previous experiment as Grid $2 \times 2$ = Ring-4.
- Grid $3 \times 2$: 75 success, 25 failure.
- Grid $4 \times 2$: 83 success, 17 failure.
- Grid $5 \times 2$: 79 success, 21 failure.
- Grid $6 \times 2$: 79 success, 21 failure.

**Interpretation:** In comparison with the previous experiment, we have added several new possibilities of communication, namely those edges which transform the rings into grids, but obviously without a significant effect to the distribution of success and failure. Also in this experiment, the average success rate seem to be restricted to $\approx 80\%$, perhaps even slightly worse that in the previous experiment.[6] Also in this case, the assumed increase of success has not turned out to be very fast in terms of the increasing number of nodes in the graph. A comparison between the results of this experiment and the previous one is shown in **Fig.18**.

[6]Similar contra-intuitive results are known from the area of traffic flow simulation, where the insertion of new streets causes even worse traffic jams instead of improving the situation.

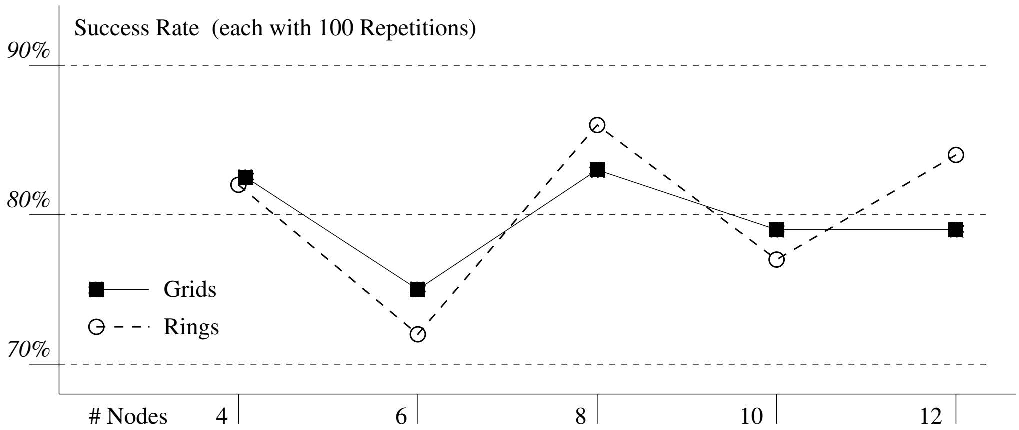


Figure 18: *comparison of SpanningTree success in rings and grids*

### 4.2.8 Messages in T-Prime Cliques

With the last experiment reported in this section, we drive our tool to the limit of its performance abilities. Here, our main question asks for the size of the graphs that our small prototype is able to handle. But by the way we also get some interesting results on the empirical average complexity (which means the number of exchanged messages) of the SpanningTree algorithm applied to its worst-case inputs: the T-prime cliques!

**Preliminaries:** To ensure that our cliques are T-prime, which is a sufficient (yet not necessary) precondition for the success of SpanningTree, we take *cliques with a prime number of nodes*, which is also a sufficient (yet not necessary) precondition for the T-prime property [10]. We take clique sizes of the first ten prime numbers, thus $2, 3, 5, 7, 11, 13, 17, 19, 23, 29$ (in number of nodes). The according numbers of edges can be computed via the formula of above. We count the number of exchanged messages $m$ until success, and we repeat the experiment 5 times per graph.

**Result:** The following numbers $m$ of exchanged messages have been counted (whereby $\hat{m}_p$ at the end of the lines are the after-computed mean-values).

- Clique-2 (with 1 edge): 5, 5, 5, 5, 11 $\hat{m}_2 \approx 6.2$
- Clique-3 (with 3 edges): 14, 21, 15, 17, 12 $\hat{m}_3 \approx 15.8$
- Clique-5 (with 10 edges): 64, 34, 37, 61, 60 $\hat{m}_5 \approx 51.2$
- Clique-7 (with 21 edges): 177, 91, 133, 89, 101 $\hat{m}_7 \approx 118.2$
- Clique-11 (with 55 edges): 337, 331, 457, 376, 348 $\hat{m}_{11} \approx 369.8$

- Clique-13 (with 78 edges): 504, 540, 629, 581, 594 $\hat{m}_{13} \approx 569.6$
- Clique-17 (with 136 edges): $973, 1215, 1720, 1258, 1409$ $\hat{m}_{17} \approx 1315.0$
- Clique-19 (with 171 edges): $2403, 1425, 1829, 1960, 1708$ $\hat{m}_{19} \approx 1720.4$
- Clique-23 (with 253 edges): $2403, 3637, 3064, 2858, 2557$ $\hat{m}_{23} \approx 2903.8$
- Clique-29 (with 406 edges): $3869, 4037, 3131, 4363, 2879$ $\hat{m}_{29} \approx 3659.4$

**Interpretation:** Similar to the earlier experiment wherein we had counted the messages until success for the ElectionNet algorithm, we now observe a rapidly increasing number of exchanged messages also for the SpanningTree algorithm, too. However we also can see the SpanningTree algorithm exchanges less messages than ElectionNet. Here, as depicted in **Fig.19**, we only have $\mathcal{O}(m) \approx 10 \cdot \mathcal{O}(|E|)$ where $E$ is the set of edges of the graph.

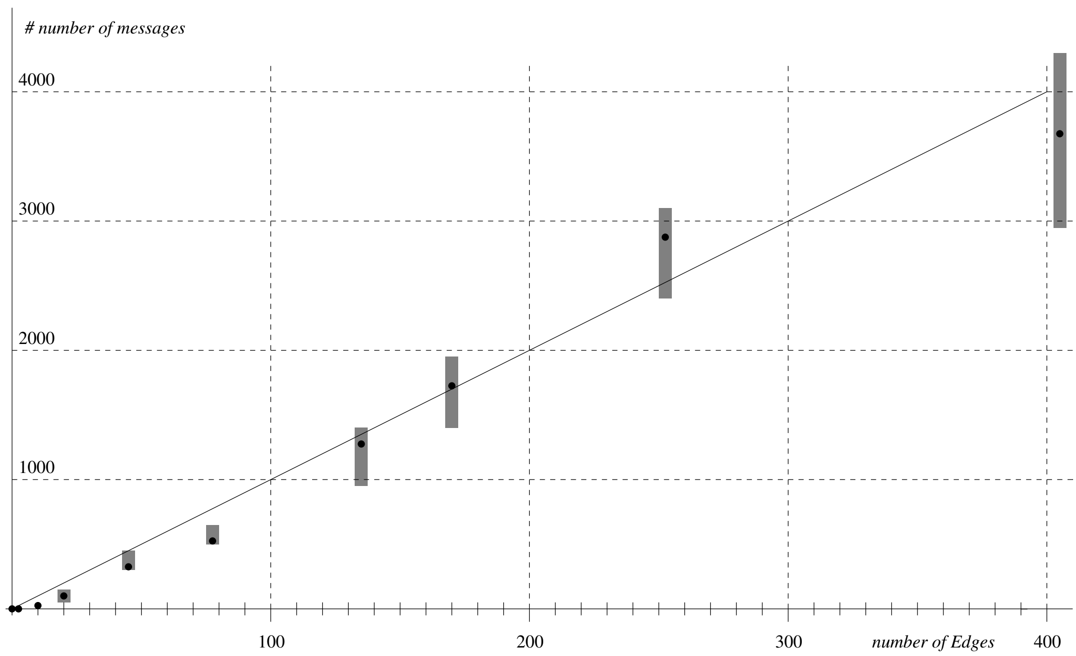


Figure 19: *linear average behaviour of SpanningTree on T-prime cliques*

This result might indicate that we better should use the SpanningTree algorithm as an election algorithm (with the root of the spanning tree as the elected node) at least in the case of T-prime graphs. However one should also keep in mind that the real-time performance of a distributed algorithm does not only depend on the number of exchanged messages during the computation. In fact, the simulative computations in this experiment have consumed considerably more real-time than in the ElectionNet experiment, though less messages have been exchanged here. It would certainly be worth an extra discussion on the question if the number of exchanged messages can be an adequate complexity measure at all. This question, however, leads us beyond the scope of this report.

The current JAVA implementation of our $\alpha$-version prototype has certainly its limits with respect to the computation speed on larger input graphs. It might be the case that another programming language will be chosen for one of the following releases of our tool.

In the SpanningTree algorithm, as reported in [10], the in-tree communication flows only along the branches of the tree currently under computation. Perhaps the performance of this algorithm could be improved if also the other edges of the graph were used for in-tree communication.

## 4.3 Section Summary

In this section we have reported on several experiments which have been made in order to explore the behaviour of our tool prototype together with some distributed algorithms as described by Metivier. Though the distributed computing is only simulated one can observe some interesting behavioural properties of the tested distributed algorithms.

Moreover it has turned out that our tool prototype needs to be improved under three points of view. First, no support for fast repetition is given at the moment such that long series of experiments turn out to be very tedious. Second, only small graphs can be handled by the tool at the moment. Third, the JAVA language is perhaps not the very best implementation language for a high-performance tool behaviour.

Nevertheless we seem to have shown that our concept of providing visualization support for distributed computations is feasible and worthwhile to be kept and improved in future.

# 5 The Refined Model

After having given the “look and feel” of our tool in the previous sections, we now have to provide a more sophisticated model in order to better describe those phenomena. For this purpose, we introduce a special node, the *visor*, which represents an entity that can visualize the changes of states in a network performing distributed computation. Thus the visor is a conceptual abstraction of the GUI component of our actual prototype. Moreover, a notion of partially ordered *time* is introduced which is necessary to understand the behaviour of our tool as described in the section 4 and further discussed in section 6.

## 5.1 The Visor

The visor (observer) is a special node $V$ which is connected by an edge $x_i$ to every other node $p_i$ in the graph $G$. The visor maintains an image $img(G \backslash V)$ of the whole graph (except of the visor itself) which can be displayed on a screen. As soon as the internal state $s$ of a node $p_i \in G$ has changed to $s'$ due to distributed computation, $p_i$ sends a state-change-message $m_{s \to s'}$ to $V$ via $x_i$, and $v$ displays an updated image $img(G \backslash V)'$ on the screen. In **Fig.20** the situation is sketched.

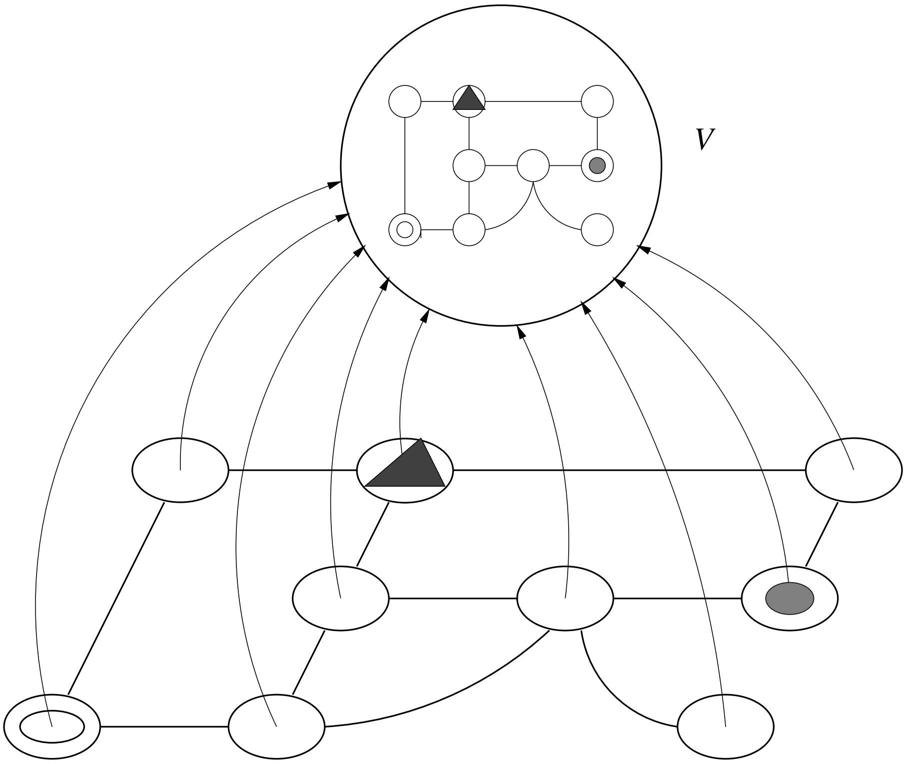


Figure 20: *visor node displaying the current state of the rest of the graph*

**Communication:** Please note that, in principle, communication between a node $p_i$ and the visor $V$ takes place in the same way as between two neighboured nodes $p_i$ and $p_j$. In our prototype implementation (see section 3), we have indeed some bi-directional communication between $V$ and the other nodes $p_i$ for reasons of efficiency. For the ideal case, however, we have to assume that the visor has *no influence* on the events in $G \backslash V$ during a distributed computation which means that $V$ can *only receive* messages but cannot send anything back into the net. In Fig.20, this model assumption is sketched by the tiny arrows with destination in $V$.

## 5.2 Partially ordered Time

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.

**Assumption:** *There is no common clock.*

**Justification:** Assuming anonymous networks where each node does only know its immediate neighbours (see section 2), having a common clock would mean having a graph with at least one (clock)node connected to all other nodes. (Information can only be transmitted via the edges.) However, we cannot impose any constraint on the topological shape of our graphs a-priori. Though the visor node $V$ is a node connected to all other nodes it cannot serve as a common clock because we have already stated that $V$ is a receiver only which cannot provide any information to the other nodes. (Tertium non datur.)

**Assumption:** *The communication speed is unknown.*

**Justification:** 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 $M$.

**Assumption:** *The network geometry is unknown.*

**Justification:** 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 lengths are dependent from the hardware realization (wires) of the system which is not specified by the model. Moreover, we do not know much about the velocity of travelling messages in the net. As a consequence, we may not even assume the triangle-inequation of the Euklidian geometry, as sketched in **Fig.21**.

**Consequences:** Each node $p_i$ in our network has got its *private time* relation $\prec_i$. Let us assume further that each two nodes $p_i \neq p_j$ can be connected by at most one FIFO communication link (as described in section 2). Moreover, let $t_i(E)$ be the point of time of some event $E$ in the time scale $\prec_i$ of a node

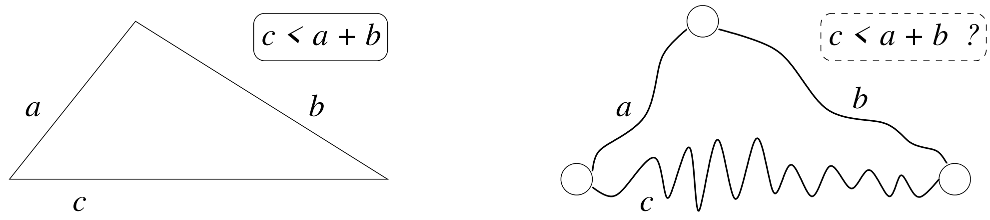


Figure 21: *Euklidian geometry may not be assumed*

$p_i$, and let $m_1, m_2$ two messages, each from a node $p_i$ to node $p_y$. Finally, let $E1x, E2x, E1y, E2y$ the following events. $E1x$: "$m_1$ departs from $p_x$". $E2x$: "$m_2$ departs from $p_x$". $E1y$: "$m_1$ arrives at $p_y$". $E2y$: "$m_2$ arrives at $p_y$". Then we can know nothing else but the following:

- $t_x(E1x) \prec_x t_x(E2x) \Longleftrightarrow t_y(E1y) \prec_y t_y(E2y)$
- $0 < v(m_1) < \infty \Longleftrightarrow t_x(E1x) \prec_x t_x(E1y)$
- $0 < v(m_1) < \infty \Longleftrightarrow t_y(E1x) \prec_y t_y(E1y)$
- $0 < v(m_2) < \infty \Longleftrightarrow t_x(E2x) \prec_x t_x(E2y)$
- $0 < v(m_2) < \infty \Longleftrightarrow t_y(E2x) \prec_y t_y(E2y)$

whereby $t_i(.)$ is a point of time in the local system of node $p_i$. Temporal relationships between $E1x$ and $E2y$ or between $E1y$ and $E2x$ are undefined. Moreover, if a processor $p_x$ sends a message $m_1$ to its neighbour $p_y$ and another message $m_2$ to another neighbour $p_z$ such that $t_x(E1x) \prec_x t_x(E2x)$, nothing is known about the relationship of the arrivals $E1y$ and $E2z$ in terms of the time scales $\prec_x$, $\prec_y$, $\prec_z$ (please remember Fig.21). Thus: a total order of events in the network is impossible from a local point of view [5].

These minimal properties of our model make it difficult for the visor to display a logically consistent "movie" of the state changes during a distributed computation — a problem which is discussed in section 6.

## 5.3 Section Summary

The visor is a special node in the network representing the state of all other nodes in the network. As the visor can only receive messages, its operations do not disturb the distributed computation in the rest of the network. Each node in the network has its private time scale and does not know much of the temporal relation of events within the other nodes of the network. A total order of events is impossible from a local point of view, which makes it difficult for the visor to display a consistent "movie" of the events occurring in the network during a distributed computation.

# 6 Consistency of Visualization?

With the terms introduced in section 5, we are now able to understand a certain *transient inconsistency phenomenon* which sometimes occurs during our experiments (as described in section 4). We explain the reason why we sometimes observe such transient inconsistency, and we present a simple solution to this problem (which is, however, not yet implemented in our tool).

## 6.1 Motivation: Pseudo Deadlocks

In [10] it is explained that the distributed spanning tree algorithm may possibly run into a deadlock when applied to a *non*-T-prime graph. Indeed, when simulating such a (real) deadlock situation with our tool, the visor correctly displays the deadlock as the final image of the distributed computation "movie". Sometimes however, as sketched in **Fig.22**, the visor displays the image of a deadlock which is overwritten by a subsequent image of the ongoing distributed computation. We call such a transient inconsistency phenomenon a *pseudo deadlock* which could be observed rather often during our SpanningTree experiments. Because a valid spanning tree is eventually computed in our example —see Fig.22*d*—, we know that the deadlock-like situation displayed in Fig.22*c* can never have happened in reality during this distributed spanning tree computation.

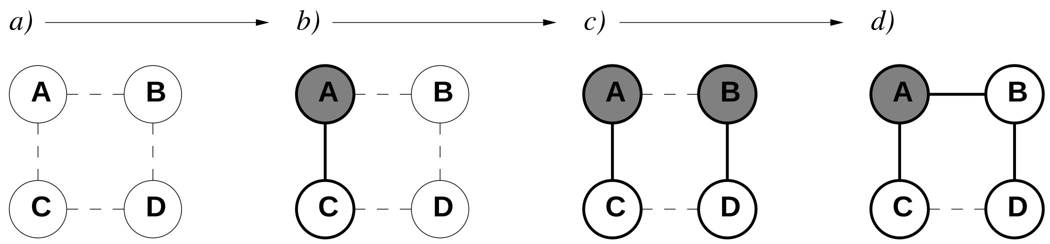


Figure 22: *pseudo deadlock (c) displayed during a spanning tree computation*

**How Pseudo Deadlocks occur:** In section 5 it has already been explained that the visor node cannot rely on a global clock in order to determine when a certain event (to be displayed) has been taken place. Moreover: without any concept of *causality* the visor has no possibility of inferring a proper order of the events to be displayed. Such a "silly" visor (which does not even need any memory) can display the occurring events only in its local temporal order which is completely dependent from the unknown velocity of the received messages. This is demonstrated in the following.

Let us assume now that the event-report behaviour of each node in the net is *reliable* which means that each two local state change events are reported to the visor in the same order as the events themselves, thus:

$$E1 \prec_x E2 \Longleftrightarrow rep(E1) \prec_x rep(E2).$$

However, even under this good circumstances the bad movie of Fig.22 can be constructed easily as shown in **Fig.23**. Without any notion of causality such a bad construction is always possible because the physical message velocities as well as the physical distances between the nodes can vary almost ad-libitum according to the assumptions of section 5.

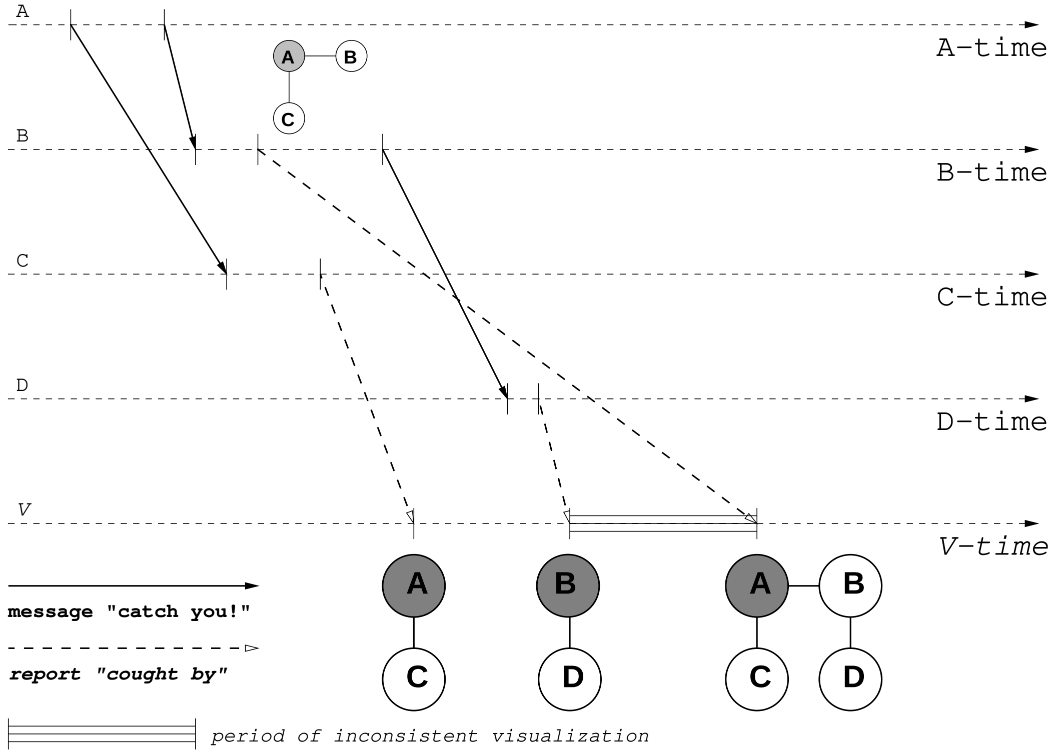


Figure 23: *how to make a bad movie out of a good story*

Having demonstrated the reasons of transient inconsistencies of visualization by the example of pseudo deadlocks in the context of distributed spanning tree computations according to [10], we will now discuss two ways of “teaching” our silly visor for avoiding the display of inconsistent movies of distributed computations.

## 6.2 The Censor Solution

Let $\mathcal{I} = <I_0, \ldots, I_T>$ a finite sequence of images of a terminating distributed computation displayed by the visor $V$. The initial image $I_0$ and the terminal image

$I_T$ must be shown anyway, because they represent the *operational semantics* of the underlying distributed computation. Within the subsequence $<I_1, \ldots, I_{T-1}>$, however, a *censor* (residing in $V$) could suppress transient inconsistencies as follows.

**Procedure:** Let $I_i$ be the image currently displayed by the visor. Then, $I_{i+1}$ and $I_{i+2}$ are constructed (according to the next incoming state change reports) but not yet displayed. If the sequence $<I_i, I_{i+1}, I_{i+2}>$ is consistent, $I_{i+1}$ becomes the image currently displayed by $V$. Otherwise, $I_{i+1}$ is suppressed (deleted) by the censor, and image $I_{i+2}$ is now put into the delayed intermediate role by pushing $I_i$ at the place of the suppressed image $I_{i+1}$. **Fig.24** sketches the main part of the censor algorithm.

```
i = 0;
display = I(i);
while i < T repeat
   {fetch I(i+1);
    fetch I(i+2);
    if consistent(I(i),I(i+1),I(i+2))
      {display = I(i+1);
       let i = i+1;}
   else
      {let I(i+1) = I(i);
       let i = i+1;};};
display = I(T);
```

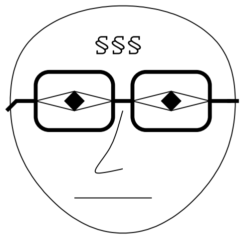


Figure 24: *how a censor can suppress inconsistencies of visualization*

**Discussion:** At least the following advantages (+) and disadvantages (−) of the censor solution can be mentioned:

+ It is easy to understand, because no notion of causality is required. The question of how the events arise in the network of distributed computation is completely irrelevant to the censor.

+ A constant image memory of size $\mathcal{O}(I_x)$ is sufficient for the censor to work.

− It is very likely that we will not see the complete movie of a distributed computation, but only a movie with certain blackouts. In the worstcase, we will only see the $I_0$ and $I_T$ while the whole rest of the movie $<I_1, \ldots, I_{T-1}>$ is completely black.

– The notion of inconsistency is a *semantical* one which is not syntactically coded in the "brain" of the censor. This means that for each kind of distributed computation which shall be displayed by $V$ a new censor with an according problem-specific knowledge of inconsistency has to be introduced.

## 6.3 The Causality Solution

In our "everydays' psychology" [7], the notion of causal dependency seems closely related to our notion of temporal sequentiality. But as we cannot rely on a common clock in our model of distributed computing, we need other means which support the visor node $V$ to display the occuring state change events in a proper order. For this purpose we introduce *dependency stamps* in $\mathbb{N}$ which are a slight modification of the well-known concept of relative clocks by Leslie Lamport [5].

The idea of our stamps is that two different events $E_i$ and $E_j$ stamped with the same number $n \in \mathbb{N}$ are independent from each other and may therefore be displayed by $V$ in any order (with respect to $\prec_V$). On the other hand: when $E_i$ and $E_j$ are in a (direct or indirect) causal dependency relation, then their dependency stamps are different from each other, and the event with the minor stamp must be displayed by the visor first. Please note the possibility that independent events may also carry different dependency stamps which, however, does not hurt at all the correctness of the method.

**Stamped Messaging:** Altogether, the following simple concepts (axioms) are sufficient for displaying the events occuring in the history of a distributed computation in a proper order:

- Each message $m$ travelling through the net is augmented to a tuple $<m, n>$ by an integer number $n \in \mathbb{N}$.
- Each node $p_i$ is provided with an internal message register $\mu_i$ of type $\mathbb{N}$. Initially we have $\mu_i := 0$.
- On each arrival of a message $<m, n>$ at $p_i$, the content of the according message register is updated $\mu_i :\rightarrow \mu_i + n + 1$.
- When sending a message $m$ from $p_i$, this message is augmented with the current content of the message register to $< m, \mu_i >$. A node may send internal messages to itself.
- When terminating its (local) computations, a net node sends a final message $<\hat{m}, \top>$ to the visor node $V$, whereby $n :< \top \; \forall n \in \mathbb{N}$.

[7] which seems to be a result of the evolution of brain in the history of nature

- Each internal state is reported. Thus: Be $s \neq s'$ two different internal states of a node $p_i$. Then we have: $t_i(s) \prec_i t_i(s') \Longrightarrow \exists$ report $<m, n>$ from $p_i$ to the visor $V$ such that $t_i(s) \prec_i t_i(send <m, n>) \prec_i t_i(s')$.

- State changes are induced by received messages. Thus: $t_i(s) \prec_i t_i(s')$ with $s \neq s'$ for some node $p_i \Longrightarrow \exists$ message $<m, n>$ from anywhere to $p_i$ such that $t_i(s) \prec_i t_i(rec <m, n>) \prec_i t_i(s')$.

- Each two messages sent from a node with the same interal state carry the same flag: Be $t_i(s) \prec_i t_i(send <m, n>) \prec_i t_i(send <m', n'>) \prec_i t_i(s')$ for a net node $p_i$ such that $\nexists$ internal state $\sigma$ with $t_i(s) \prec_i t_i(\sigma) \prec_i t_i(s')$ with $s \neq \sigma$ or $\sigma \neq s'$. Then we have: $s = s' \Longleftrightarrow n = n'$.

- When the visor $V$ receives a message $<m, n>$ the display of the reported event $E_{\lhd m,n \rhd}$ is delayed until $V$ is shure that the receive of any other message $<m', k>$ with $k < n$ is impossible.

**Example:** In **Fig.25** it is shown how some net node $P$ may receive and send some dependency-stamped messages. The register is incremented at every event, and the numbers represent the "flow of causality". Please note that both the two messages sent away with stamp "5" are assumed to be independent, because no other message has arrived in the meantime which could have increased the dependency level of node $P$. Hereby we assume (without loss of generality) that the increase of dependency can only be mediated by the arrival of messages. Thus: in our model, self-dependencies have to be expressed by internal messages of a node to itself, as also depicted in the example of Fig.25.

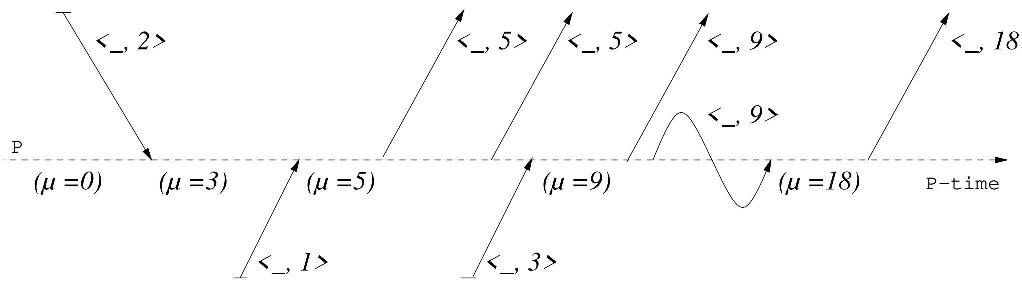


Figure 25: *receiving and sending dependency-stamped messages*

In the following we show that it is possible for the visor node $V$ to recognize and to obey the dependency of events that arise during a distributed computation in the rest of the net. For this purpose we (completely) distinguish dependent events *on one and the same* node from dependent events on *different* nodes. When the visor obeys these event dependencies, the visualization of the distributed computation is called *dependency-consistent*.

**Definition:** Let $p_i$ a node and $\prec_i$ its time relation. Two different points of time $t_i(.) \prec_i t_i(.')$ are called *almost at the same time*, denoted as $t_i(.) \approx_i t_i(.')$, if there does not exist any message $<m, \_>$ such that $t_i(.) \prec_i t_i(send <m, \_>) \prec_i t_i(.')$ or $t_i(.) \prec_i t_i(rec <m, \_>) \prec_i t_i(.')$ •

### 6.3.1 Lemma: $\mu_i$ as Local State Indicator

Let $p_i$ a node and $s \neq s'$ two internal states of it. Let further be $k, k' \in \mathbb{N}$ two states of the message register such that $t_i(s) \approx_i t_i(\mu_i = n) \prec_i t_i(s') \approx_i t_i(\mu_i = n')$. Then we have $n < n'$.

**Proof:** Because $s \neq s'$, there is received a message $<k, \kappa>$
with $t_i(s) \prec_i t_i(rec <k, \kappa>) \prec_i t_i(s')$.
Because of $t_i(s) \approx_i t_i(\mu_i = n)$ and $t_i(s') \approx_i t_i(\mu_i = n')$
we also have $t_i(\mu_i = n) \prec_i t_i(rec <k, \kappa>) \prec_i t_i(\mu_i = n')$.
Therefore we have $n' = n + \kappa + 1 + x$, with $\kappa \geq 0$.
In case that no further messages are received between $s$ and $s'$ we have $x = 0$, otherwise $x \geq 0$ according to the register increment on message receive. $\square$

**Comment:** Because temporal dependency covers causal dependency (but not vice versa), we have caught all causal dependencies by catching the temporal ones.

### 6.3.2 Lemma: Trans-Node-Dependencies and $\mu$-Values

Let $p_i, p_j$ two nodes in the net with internal states $s, s'$ and message registers $\mu_i, \mu_j$ such that $t_i(s) \approx_i t_i(\mu_i = n)$ and $t_j(s') \approx_j t_j(\mu_j = n')$. Then the following proposition holds: If $s'$ is causally dependent from $s$ (denoted as $s \rightharpoondown s'$), then $n' > n$.

**Proof:** (*Preliminary*, not proven here: Causal dependency is obviously transitive such that any indirect causal dependency $x \rightharpoondown y \rightharpoondown z$ may be reduced to a direct dependency $x \rightharpoondown z$.)

Because $s \rightharpoondown s'$ and $t_i(s) \approx_i t_i(\mu_i = n)$ and $t_j(s') \approx_j t_j(\mu_j = n')$, there exists a message $<k, \kappa>$ from $p_i$ to $p_j$ such that $t_i(\mu_i = n) \prec_i t_i(send <k, \kappa>)$ and $t_j(rec <k, \kappa>) \prec_j t_j(\mu_j = n')$, whereby $\kappa = \kappa' + 1$. Therefore we have $n < \kappa \leq n'$.
$\square$

**Comment:** Please note how this lemma depends on the previous one. Please note further that we cannot speak of temporal dependencies in this lemma, because the time systems of $p_i$ and $p_j$ are not comparable to each other.

### 6.3.3 Lemma: Dependency-Consistent Visualization

Let $<m,n>$ be a message arriving at the Visor $V$, reporting some state change event $E_{\lhd m,n \rhd}$ which has occured in the net, and let $D_{\lhd m,n \rhd}$ the event of displaying an Image of $E_{\lhd m,n \rhd}$ by $V$.

The visualisation provided by $V$ is dependency-consistent if for any two messages $<m,n>$ and $<m',n'>$ arriving at $V$ (during a distributed computation) with $n' > n$ and $t_V(rec <m',n'>) \prec_V t_V(rec <m,n>)$ their visualisation is delayed and inverted such that

$t_V(rec <m',n'>) \prec_V t_V(rec <m,n>) \prec_V t_V(D_{\lhd m,n \rhd}) \prec_V t_V(D_{\lhd m',n' \rhd})$.

**Proof:** If $E_{\lhd m,n \rhd} \rightharpoondown E_{\lhd m',n' \rhd}$ (causal dependency) then, by the previous lemmas, $n' > n$. Temporal dependency, assured by $t_V(D_{\lhd m,n \rhd}) \prec_V t_V(D_{\lhd m',n' \rhd})$, obviously covers the causal dependency. $\Box$

**Comment:** After the previous explanations, this result is not very surprising. In order to finish the feasibility proof of our visualisation concept, it remains only to show that the delay of visualisation —sufficient amount of memory presumed— can be applied without further problems, which is the statement of the following and last lemma of this section.

### 6.3.4 Lemma: Finite Visualization Delay

Let $C$ be a terminating distributed computation in a network $G$. Let $<m,n>$ be a message arriving at the Visor $V$, reporting some state change event $E_{\lhd m,n \rhd}$ which has occured in $C$, and let $D_{\lhd m,n \rhd}$ the event of displaying an Image of $E_{\lhd m,n \rhd}$. Then $t_V(D_{\lhd m,n \rhd}) \prec_V \infty$.

**Proof:** Let $P = \{p_1, \ldots, p_i, \ldots, p_k\} \in G$ the nodes of the network, and let $p_i$ the node from which the message $<m,n>$ has arrived at $V$. Per definitionem $V$ has knowledge of $P$. $E_{\lhd m,n \rhd}$ is displayed as soon as $\forall p_j \in P$ with $p_j \neq p_i \; \exists <m_j, n_j>_j$ with $0 \prec_j t_j(send <m_j, n_j>_j)$ and $t_V(rec <m,n>) \prec_V t_V(rec <m_j,n_j>_j)$ such that $n_j \geq n \; \forall j = 1, \ldots, k$ with $j \neq i$. Because of $|P| < \infty$ and $C$ terminating and $\top > \mathbb{N}$ for every termination report $<\hat{m}, \top>_j$, the point of time $t_V(D_{\lhd m,n \rhd})$ exists. $\Box$

**Comment:** Please note that our concept of delay must not be understood as absolute, but relative to the events in the network. Of course, if the network would stop reporting its events to the visor because of some reason, one cannot expect the visor to display a finite history of those unreported events.

## 6.4 Section Summary

Because our tool prototype is still in an early stage of implementation, it may display the events of a (simulated) distributed computation with transient visualization inconsistencies. The pseudo deadlocks sometimes displayed by our tool during the distributed computation of spanning trees (following the algorithm of Metivier) are special examples of such transient visualisation inconsistencies.

In order to overcome those drawbacks in a future version of our tool, two concepts of implementing consistent visualisation have been introduced and discussed:

- **The Censor Solution:** This approach does only require a constant amount of memory for $V$, but vizualisation blackouts may the price of the technical primitivity of that concept. Moreover: The idea of consistency is not syntactically defined in this concept, but semantically and problem-specific, such that a specific censor would have to be plugged into the system for the execution of any different distributed algorithm. The censor solution might often be useful when only the occurance of transient visualization inconsistencies (but not their detailed properties) shall be investigated. Instead of suppressing the inconsistent images, the censor could also attach a special flag to them such that the amount of visualization inconsistencies during a distributed computation could be counted.

- **The Causality Solution:** In this approach to consistent visualization, the notion of consistency is characterized by covering causal dependency (in the net) with temporal dependency (in the visualization component $V$). No visualization blackouts will be witnessed with this concept, but unlimited memory (in principle) is necessary to cope with arbitrary visualization delays. Nevertheless, those delays are finite as long as the termination of the underlying distributed computation is guaranteed. The employed method of monotoneously numbering the occuring messages in $\mathbb{N}$ is a slight modification of the well-known method of Lamport's relative clocks.

It might be useful to implement not only one of these, but both approaches in a future version of our tool.

# 7 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 [11].

- 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 [6] 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.

# 8 Conclusion

In this report, we have presented the current state of our work on the visualization of distributed algorithms. However, future work remains to improve our tool particularly to handle huge graphs as well as real-world networks. Parts of the tool are under re-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. We think that our tool is useful for educational purposes in order to explain the execution of distributed algorithms, and also for researchers in distributed algorithms who require tools for tests and experiments.

## Acknowledgments

Many thanks to *Yves Métivier* who has patiently answered several dozens of questions. Thanks to our students who have implemented our tool in their semester project. The TMR network GETGRATS [22] has granted the research activities of one of the authors at the LABRI Institute. Last but not least, the IFBI Dagstuhl [23] has given financial support for travelling for that the contents of this report could be presented at a Dagstuhl-Seminar in October 2000.

# 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] Michael Goedicke & Gabriele Taentzer & Torsten Meyer: *View-Point-oriented Software Development by Distributed Graph Transformation.* Proceedings of the Requirements Engineering'99 Conference, Limerick 1999. Available at `http://tfs.cs.tu-berlin.de/~gabi/gGMT99.ps.gz`

[5] 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

[6] 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

[7] Yves Métivier & Igor Litovsky & Éric Sopena: *Graph Relabelling Systems and Distributed Algorithms.* In G.Rozenberg et al. (Eds.): Handbook of Graph Transformation 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`

[8] 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`

[9] Yves Métivier & Nasser Saheb & Akka Zemmari: *Randomized Rendez-Vous.* LaBRI Report 1228-00, Université Bordeaux, January 2000. Available by `yves.metivier@labri.u-bordeaux.fr`

[10] Yves Métivier & Pierre-A. Wacrenier: *A distributed Algorithm for Computing a Spanning Tree in Anonymous T-Prime Graphs.* LaBRI Report 1229-00, Université Bordeaux, January 2000. Available by `yves.metivier@labri.u-bordeaux.fr`

[11] 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

[12] Armando Rocha: *Neural Nets — A Theory for Brains and Machines.* Lecture Notes in Computer Science 638, Springer-Verlag (Berlin), 1992. Available at `http://www.springer.de/comp/lncs/`

[13] Wolf Singer: *Das Gehirn verarbeitet Informationen massiv parallel.* Spektrum der Wissenschaft, Vol. Spezial 1/2000, ISSN 0943-7096 (Heidelberg), 2000. Available at `http://www.spektrum.de/sonderhefte.html`

[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

[20] C. Touzet: *Reseaux de Neurones Artificiels — Introduction au Connexionnisme.* EC2 Publisher (Nanterre), 1992

[21] Andrew Wuensche & Mike Lesser: *The Global Dynamics of Cellular Automata.* IBSN 0-201-55740-1, Addison Wesley, 1992

[22] GETGRATS: `http://www.di.unipi.it/~andrea/getgrats/`

[23] IBFI Dagstuhl: `http://www.dagstuhl.de/`