Int J Softw Tools Technol Transfer (2006) 8: 77–91 / Digital Object Identifier (DOI) 10.1007/s10009-005-0198-5

Interactive visualization of large state spaces

Jan Friso Groote, Frank van Ham

Department of Computer Science, Technische Universiteit Eindhoven, P.O.Box 513, 5600 MB, Eindhoven, The Netherlands
e-mail: {jfg,fvham}@win.tue.nl

Published online: 14 October 2005 – © Springer-Verlag 2005

Abstract. Insight into the global structure of a state space is of great help in the analysis of the underlying process. We advocate the use of visualization for this purpose and present a method to visualize the structure of very large state spaces with millions of nodes. The method uses a clustering based on an equivalence relation to obtain a simplified representation, which is used as a backbone for the display of the entire state space. With this visualization we are able to answer questions about the global structure of a state space that cannot easily be answered by conventional methods. We show this by presenting a number of visualizations of real-world protocols.

Keywords: State space – Visualization – State transition graphs – Transition systems – Graph drawing

1 Introduction

In the last decade, advances in computer hardware and software have made it possible to effectively analyze the behavior of complex software systems by means of state transition systems. Techniques based on explicit state enumeration can now deal with systems consisting of several million states and have reached a scale at which they become sufficiently effective to assist in the design and testing of real-world software systems. However, they also confront us with state transition graphs [1] of enormous dimensions, of which the internal structure is generally a mystery.

The most common approach to obtaining insight into the structure of large state spaces is by abstracting from actions and/or state information and by reducing the state space modulo a suitable equivalence. Suitable equivalences are trace, weak, branching, or strong bisimulation, for example. Although this approach maintains the behavioral aspect of state spaces, it destroys their structure. Typical questions that cannot be answered by using such techniques are:

- How many states are in each phase of a protocol?
- How many loosely connected parts, i.e., parts with few paths between them, does the state space have? This question might be relevant to determine the effectiveness of different testing approaches.
- Are there hot spots, i.e., groups of states that are visited relatively often when randomly traversing the state space? Are there parts of the state space that have extremely low probability to be visited?

On top of that, in many cases the reductionist approach mentioned above does not help in getting insight into the behavioral aspect of state spaces either. This is simply because even after abstraction, state spaces are often larger than a few hundred states, which is well above the limit of current generic graph drawing packages such as dot [6] or neato [6], where each state and each transition is explicitly drawn. Other approaches such as [3, 12], although targeted specifically at finite automata, suffer from the same scalability problem.

The images these methods produce suffer from information overload, cluttering, and generally take too long to generate. A recent method by [13] is able to layout graphs of millions of nodes in a matter of minutes but is only effective for highly regular gridlike graphs.

This leaves us with analysis techniques like simulation, testing, model checking, and the verification of behavioral equivalences. Although such techniques are very suitable to answer all kinds of questions about state spaces, they do not provide insight into the structure of the state space itself.

It is important to realize the vagueness of the problem we are dealing with here. When trying to apply state space techniques to real-world cases, researchers initially do not have any insight into the structure of this state space. Not having any insight into the structure of a problem makes it very hard to formulate concrete questions about a problem in general. A visualization should therefore focus foremost on answering the question: What is the global structure of this state space? The answers to more concrete questions, such as how many deadlocks there are or how many distinct paths there are between states X and Y, can also be found by using other specialized tools and subsequently mapped to the image of the global state space.

In this paper, which is an extended version of [9], we present a number of applications of a novel technique [10] that can effectively display state transition systems consisting of millions of states and clarify their structure in this way. Essentially, the scalability of the technique is limited by the capacities of the graphics hardware. With the fast development of graphics hardware (a scene that took 20 min to render in 1998 can now be displayed at 60 Hz), our visualization technique will soon be suitable for much larger state spaces. Besides being scalable, this technique is also computationally inexpensive and very predictable (that is, local changes in structure usually do not lead to global changes in visualization). A prototype version is freely available at `http://www.win.tue.nl/~fvham/fsm/`.

Input state spaces are generated on the basis of behavioral specifications in μCRL [4] from which state spaces in SVC format [14] are produced, although the tool also accepts different formats (such as BCG) through the use of conversion tools. In these state spaces transitions are labeled with an action's name and states are labeled with a vector of data values. The basic idea underlying the visualization technique is to use a simplified representation in the form of a tree as a backbone for the entire structure. First, the state space is layered using the shortest path distance of each node to the root. Then, the tree structure is obtained by clustering sets of states in each layer such that for each set a unique path to the root is obtained. Each set of states is subsequently modeled using a disk shape in a 3D space. Finally, all disks are connected in a manner resembling cone trees [17], forming the shapes such as the ones in Figs. 8, 9, and 13. Note that visual cues such as interactive motion, color, lighting, and transparency all add strongly to the 3D perception of the shapes; the black-and-white still pictures in this print are by no means comparable to the on-screen images. After visualizing the state space as a tree-shaped 3D object we can use coloring to stress particular aspects of the state space. Typically, coloring can be induced by intrinsic properties, such as the value of the transition label or statevector, or derived properties, such as the probability to visit a state during a random walk.

In the next section we elaborate on the clustering method mentioned above. Section 3 presents visualizations of real-world examples, and we conclude in Sect. 4.

2 The visualization algorithm

As mentioned in the previous section, we are aiming for a visualization that will help us understand the global structure of the state space. More concretely, the structural aspects that we would like the visualization to preserve are:

- **Distances**: we would like the visualization to preserve the graph theoretic distance from a state to the initial state. That is, states that can be reached from the initial state in a few steps should be positioned close to the initial state and vice versa.
- **Symmetries**: since (mathematically generated) state spaces are often highly symmetrical, we would like the visualization to reflect these symmetries.
- **Size**: larger groups of nodes at equal distance from the initial state should be represented by larger visual elements.

Note that we do not claim that any visualization that addresses these points is **the** visualization of **the** global structure of the graph. In fact, creating a visualization that shows all structural properties of a large state transition graph and is invariant to algebraic reductions of the state space might well be an impossible task. We are merely trying to create a picture of a particular state transition graph that shows a number of its structural aspects. The next sections will discuss the construction of our visualization in detail.

2.1 Formal graph structure

Since it is impossible for a human to cope with pictures containing hundreds of thousands of data elements, no matter what visualization method we use, it is unavoidable to bring a state space back to a manageable number of elements if we wish to visualize it. Where conventional methods generally reduce a state space to a smaller one based on behavioral information, we do not apply any reduction and use only structural information to cluster states together. A state transition system can be defined as a 4-tuple (S, Act, T, s), where S is the set of states, Act is a set of possible actions, the relation $T \in S \times Act \times S$ is a labeled transition relation, and $s \in S$ is the system's initial state. Each state consists of a vector of parameter values. We can represent the corresponding state transition graph by a graph (V, E, s) where a node $x \in V$ represents a state and $a_{xy} \in (E \subseteq V \times V)$ represents a directed edge (or arc) between the nodes x and y. An arc exists for every transition in the corresponding state transition system.

The first step in the reduction process is to assign each individual node a nonnegative layer or rank. The most common method is to assign each node x a rank $Rank(x)$ depending on its distance from the start node, taking edge direction into account. But other ranking algorithms are also possible, such as using a similar layering that ignores edge direction. In the remainder of this paper we refer to arcs running from a low-ranked node to a higher-ranked node as arcs pointing downward and draw these accordingly. In our ranking, all connections between consecutive ranks point downward, but undesirable long up-arcs spanning more than one rank (we call these backpointers) are frequent. These backpointers tend to spoil the resulting visualization because their endpoints can potentially be very far apart, resulting in long edges, so in our application we can interactively choose to display or hide them.

In a second step we cluster nodes based on an equivalence relation. To facilitate the definition of this relation, we define a new edge set E' by slightly modifying our original edge set E. We remove all backpointers and reverse the direction of all other up-arcs in E (if any). Note that in the final visualization, we display the original edge set E, but we base the layout on E'. We shall refer to the graph induced by the vertex set V and the modified edge set E' as graph G'. In our modified edge set E', all arcs are now either down-arcs or arcs connecting equally ranked nodes. Let $D(x)$ be the set of all nodes that can be reached from node x via zero or more arcs in E. We now define two nodes x and y to be equivalent if and only if there exists a sequence $x = z_1, z_2, \dots, z_N = y$ of nodes with equal rank such that for all $1 \le i < N$ it holds that $D(z_i) \cap D(z_{i+1}) \neq \emptyset$ (Fig. 1).

We use the resulting equivalence classes as clusters and define a relation between clusters by extending the concept of rank from nodes to clusters, that is, the rank of a cluster C is equal to the rank of any node in C (since all nodes in C have equal rank). We now define a cluster C_1 to be an ancestor of a cluster C_2 iff $Rank(C_1) = Rank(C_2) - 1$ and there exists an arc in E' connecting a node in C_1 with a node in C_2. A cluster C_2 is defined to be a descendant of C_1 if and only if C_1 is an ancestor of C_2. The resulting cluster structure forms a tree with the cluster containing the start node at the root. Figure 3 shows a small input graph and its resulting backbone structure.

Of course, in a usable application the implementation of this clustering process will have to be linear. Instead of using the naive approach of storing $D(x)$ for each node x and subsequently checking for nonempty intersections, we used a recursive algorithm that clusters all nodes in linear time. We will outline the algorithm for computing the backbone tree of a graph $G = (V, E)$ below. Assume that each node n has already been given a rank $R(n)$ and the reduced graph G' has been computed. Consider the nodes v and x with a_{vx} an arc in G'. According to the definition of G', there are two cases for x, either $R(x) = R(v)$ or $R(x) = R(v) + 1$. In the first case, node x is equivalent with node v since $D(x)$ and $D(v)$ have a nonempty intersection containing at least x. So x and v are in the same cluster.

The second case is more complicated. If the ranks of v and x differ, they are not in the same cluster. However, arcs crossing ranks can induce that v has to be merged with other nodes without a direct connection with v. We therefore first compute the cluster of x. For all nodes y in the cluster of x we can state that $D(x) \cap D(y)$ is not empty. Hence, for all these nodes y we have to add all nodes w with an arc $a_{wy} \in E'$ and a rank equal to $R(v)$ to the cluster of v, since $D(v) \cap D(w)$ is not empty (Fig. 2).

These two cases form the heart of the recursive procedure `ClusterTree(v:Node,c:ClusterNode)` given in

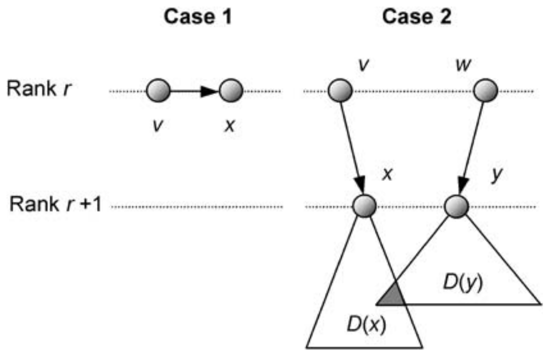


Fig. 2. Cluster algorithm case analysis

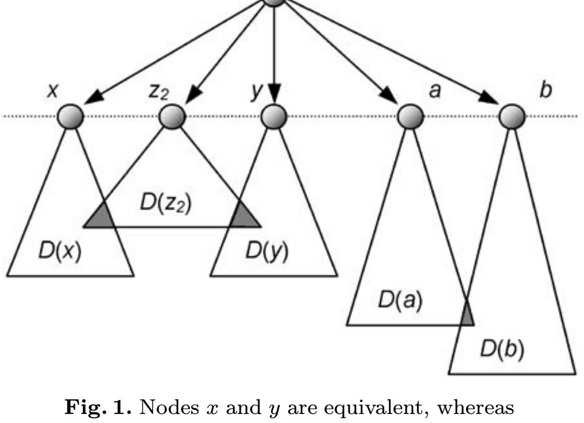


Fig. 1. Nodes x and y are equivalent, whereas nodes x and a are not

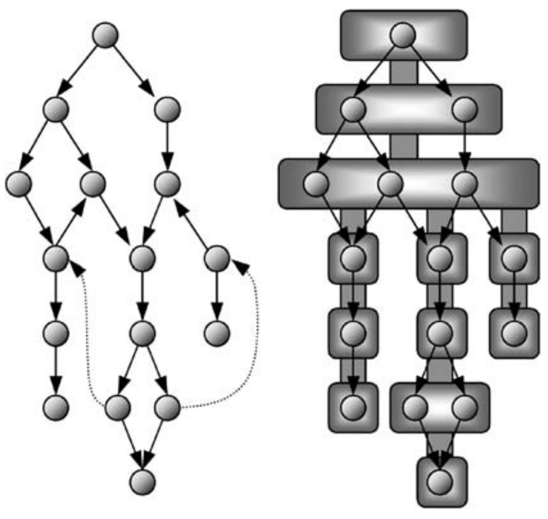


Fig. 3. Applying the cluster algorithm to a small graph

Fig. 5. The precondition of this procedure is that all nodes in `c.Nodes` are in the same cluster as node v and `v.Cluster = nil`, that is, v has not yet been assigned to a cluster. A postcondition for the procedure is that the subtree of the backbone tree with its root at the cluster of v has been fully computed. The cluster algorithm has a time complexity linear in both the number of edges and the number of vertices of G' since each edge is traveled at most once and each vertex is at most once a parameter for the procedure.

2.2 Backbone construction

Since the tree structure of clusters we obtained is much less complex than our original graph, we use it as a backbone for visualizing the entire graph. Before describing the visualization method in detail, we state our general requirements for the visualization.

– Symmetry is important, and therefore a visualization that produces a more symmetrical picture is to be favored. Clusters and nodes with the same structural properties should be treated in the same way.
– There has to be a clear visual relationship between the backbone structure and the actual graph. It is easier for the user to maintain context when inspecting a small section in detail if it looks approximately the same in closeup view as it did in the global overview.
– The size of the clusters has to be related to the number of nodes in a cluster to prevent nodes from cluttering together on the display. Clusters with a larger number of nodes have to be visualized by larger visual elements.

Although classical 2D tree layouts are very predictable, familiar, and easy to use, the lack of visualization space quickly becomes a problem when dealing with larger graphs, especially when considering that we want to visualize larger clusters as larger nodes in the tree. A popular technique to deal with this problem is to move from a 2D to a 3D layout, which gives us an extra dimension to increase the cluster size. We aim for a visualization that depicts clusters as circles in a horizontal plane. A plane is reserved for each rank, with the topmost plane containing clusters with rank 0. The backbone tree is laid out in a manner resembling cone trees [17], with ancestor clusters positioned at the apex of an imaginary cone and their descendant clusters distributed over the base of this cone. To emphasize the hierarchy in the cluster structure, truncated cones are drawn between related clusters. The overall process adheres to the basic concepts of cone trees but with a few alterations: **A**. Clusters (the nodes in the cone tree) are visualized as circles of different sizes. **B**. Symmetry is improved by also allowing clusters to be positioned in the center of the cone's base. **C**. The final resulting structure is given more "body," and extra visual cues are added.

Ad A. Normal cone trees consist of a collection of similar looking nodes. The tree nodes in our modified cone tree, however, are the clusters we defined in the previous section. Since each cluster contains a different number of nodes, we represent them by different sized circles. Nodes will be placed on the circle boundary, so we choose to keep the circle's circumference proportional to the number of nodes in the cluster, which results in the same amount of visualization space for each node.

Ad B. We present a heuristic for creating symmetrical layouts and discern the following cases for the positioning of the N descendant clusters of a cluster A:
If $N = 1$, the descendant cluster is positioned directly below A. If $N > 1$ we space the clusters evenly over the base of a cone with its apex at the center of A. The base diameter of this cone can be computed by using a recursive method similar to the one used by [5]. However, since positioning all N descendant clusters over the base may not always yield a symmetrical solution, we make the following three exceptions:

– If there is a unique largest cluster among the descendant clusters, we position this cluster directly below A in the center of the cone's base (Fig. 4a).
– If there is one unique smallest cluster among the descendant clusters, we center this cluster when there are no largest clusters centered (Fig. 4b) or when there is a largest cluster centered and the smallest cluster has no descendants. This prevents clusters from potentially overlapping each other.
– If after centering clusters based on the above exceptions only one noncentered cluster remains, we choose not to center the largest cluster. This produces a more balanced layout (Fig. 4c).

Ad C. Since nodes are positioned on the circle boundaries, most edges between nodes in a cluster and nodes in a descendant cluster will typically run within a section of space bounded by a truncated cone. A simple but effective way to reduce the visual complexity of the graph, then, is to visualize these two clusters as a truncated cone. The cone's top radius is set equal to the radius of the ancestor cluster, and the cone's bottom radius is equal to the radius of the descendant cluster. If we are dealing with multiple descendant clusters, the cone's bottom radius is equal to the radius of the base of the (imaginary) cone the clusters are positioned on and we draw the cone with higher transparency. Although this method provides a good overview of the graph's global structural

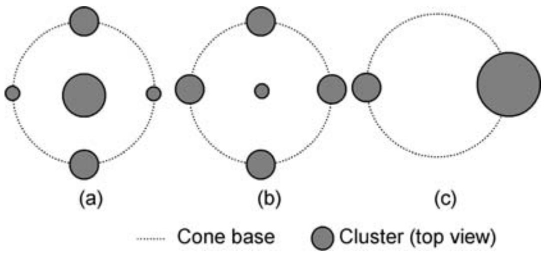


Fig. 4. Distributing clusters over cone base (*top* view)

```
type ClusterNode =
record
  Nodes : set of Node;
  Anc : ClusterNode;
  Desc : set of ClusterNode;
end;

procedure ClusterTree(v:Node,c:ClusterNode)
begin
  c.Nodes := c.Nodes ∪ {v};
  v.Cluster := c;
  forall $a_{vx}$ in $E'$ with x.Cluster=nil
    if $R(x) = R(v)$ then ClusterTree(x,c);
    if $R(x) = R(v) + 1$ then
      ClusterTree(x,new ClusterNode);
      x.Cluster.Anc := c;
      c.Desc := c.Desc ∪ {x.Cluster};
      forall y in x.Cluster.Nodes do
        forall $a_{wy}$ in $E'$ with w.Cluster=nil do
          if R(w)=R(v) then ClusterTree(w,c);
end;
ClusterTree(StartNode, new ClusterNode);
```

Fig. 5. Algorithm pseudocode

properties, it suffers from some problems inherent to 3D visualizations, the most notable being the problem of objects occluding each other. To overcome this problem and at the same time improve use of available visualization space, we rotate noncentered clusters (and their descendants) slightly outward.

2.3 Node positioning

The previous two subsections presented a method to reduce visual detail by clustering nodes and described how to visualize this reduced structure. The next step is to assign an optimal position to the individual nodes in the graph given the fact that nodes are positioned on the circle edge. An optimal positioning of nodes satisfies the following requirements:

- Short edges between nodes. A visualization is more effective if a node is kept close to its neighbors.
- Maximum possible distance between nodes in the same cluster. Nodes are to be kept as far apart as possible to reduce cluttering and may not coincide.
- Emphasis on symmetry. Where possible, emphasize symmetry in the structure by positioning nodes with the same properties in the same way.

Clearly the first two requirements contradict, since positioning two nodes in the same cluster that have the same parent node further apart leads to a greater total edge length. Another problem is the computational complexity. Although positions can be calculated by minimizing an error function or using a force-directed approach, the number of nodes we are dealing with is generally too large to provide visualization at an interactive level. We therefore select a rule-based approach in which the position of a node is governed by local structural node properties. We use a two-step heuristic to position the nodes. First, we assign initial positions, based on the positions of nodes in ancestor clusters, similar to [18]. That is, nodes are positioned at the barycenter of their parents' position. To enforce some regularity on the resulting layout, each cluster is subdivided into a number of slots, after which nodes are rounded to the nearest slot. In a second step, we adjust these positions to increase the internode distance by dividing nodes sharing a slot symmetrically over a section of the cluster. The size of this section is governed by the occupancy of neighboring slots. A more detailed description of the layout method can be found in [10].

Given the positions of the nodes, the edges between them can be visualized. The standard way to show edges is simply to draw a straight line between two nodes. Edge direction is then usually depicted with a small arrowhead, or with transparency or edge thickness. We found that such subtle cues are not effective here because of the huge number of edges. Also, the use of color is not the most intuitive cue. To show direction more effectively, we used the shape of an edge to indicate whether we are dealing with a downward or backward edge. Straight lines indicate downward edges, while curved lines denote upward edges (Fig. 8). This cue is very effective and provides a more natural way to emphasize cycles in the graph and also prevents backward edges from being obscured because they are now shown outside the cluster structure.

3 Examples

Where the previous sections were concerned with the construction of the visualization itself, this section will present the application of the technique.

3.1 Capabilities

Although the main purpose of the visualization is to give insight into the global structure of the entire state space, we can also use the image of the state space to make statements on specific aspects of the state space. Currently the tool can show:

- **(a) Symmetries**: Since the layout method maintains symmetries in the input graph, symmetries in the graph can be spotted by simply looking for symmetrical sections in the image.
- **(b) Parallelism**: During parallel execution of processes the number of possible program states expands rapidly. When more and more processes finish execution, the number of possible states decreases over time (see also Fig. 6). This behavior can generally be spotted by looking for expanding and contracting bulks in the image. Figure 9 shows that iterative parallel delivery of data causes the many consecutive bulges that can be observed, especially in the figures for five and seven jacks.

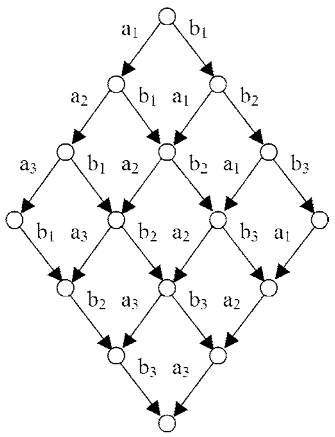


Fig. 6. Parallel composition of linear processes $a_1 \cdot a_2 \cdot a_3$ and $b_1 \cdot b_2 \cdot b_3$ illustrating expansion in the number of possible states

- **(c) Probabilities**: When given probabilistic information on each transition, we can assign a probability to each state by simulating a random walk originating at the initial state. If we have no probabilistic information, we can distribute probabilities evenly over the outgoing edges of each node. We can then use the global overview of the graph to visualize this information by coloring the clusters based on average probability, with high-probability sections in a darker color, helping us to spot live locks for example.
- **(d) Deadlocks**: We can easily mark deadlocks by having the system color clusters that contain deadlocks. Deadlocks can be identified in linear time by the system by checking all states in the graph.
- **(e) Initialization phases**: Initialization phases can be spotted by visualizing all backpointers and checking whether some of them return to the initial state. If they do not, this means that the initial state cannot be reached for a second time after execution commences.
- **(f) Semantics**: To allow a user to relate the image of the state space to the specification of that state space, we can mark states that have a specific value for a state parameter. This allows the user to pin behavioral aspects to different sections of the graph. To keep the user from having to select each possible state parameter/value combination to see which one is the best match for a given section, we also implemented a reverse mapping. That is, the user can select a specific section, and the tool gives a list of parameter value pairs with the best correlation.

The rest of this section contains some examples of situations in which a visualization of a finite automaton provides useful information that would be very hard to extract using conventional methods. Where possible, we refer back to the list above to indicate which tool capabilities we used to make a given observation. The first example consists of two simple, similar automata and is meant to illustrate the algorithm outlined in the previous section. The second and third are examples of real-world industrial protocols. The last paragraph describes some observations that we made when applying this method to systems in the VLTS Benchmark [7] set.

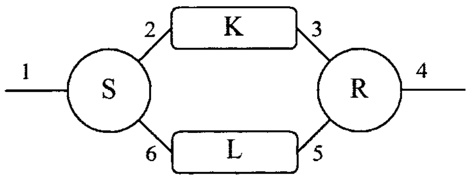
3.2 The alternating bit protocol and the PAR protocol

The alternating bit protocol (ABP) is a communication protocol concerned with data transmission between a sender S and a receiver R over unreliable channels K and L [2]. When sending a datum d, S appends a bit 0 to the datum and sends it to R over K. Upon correct receipt of this datum, R sends an acknowledgement bit 0 back to S over L. S then transmits a new datum with bit 1 appended if the acknowledgement is correctly received or resends the original datum with bit 0 if not. Both channels K and L can corrupt the data, resulting in the delivery of an error indication $\perp$ instead of the original datum. The formal specification of the ABP is given in Table 1.

In this section we present visualizations of the ABP and one of its variations, the positive acknowledgement with retransmission (PAR) protocol [16]. Figure 7 shows different visualizations of models of both protocols using two different data elements in the transmission. Individual states are depicted as spheres, transitions between states as lines, while backpointers as arcs. A truncated cone between a cluster and its multiple descendants is rendered more transparent and shows up lighter than the other cones. The start node is always located at the top of the visualization.

All visualizations depicted in Fig. 7 show a high degree of left–right symmetry (a) in the vertical axis, since

Table 1. ABP formal specification and schematic

Sender
$S = S0 \cdot S1 \cdot S$ $Sn = \sum_{d \in D} r_1(d) \cdot Sn_d$ $Sn_d = s_2(dn) \cdot Tn_d$ $Tn_d = (r_6(1-n) + r_6(\perp)) \cdot Sn_d + r_6(n)$
Receiver
$R = R1 \cdot R0 \cdot R$ $Rn = (\sum_{d \in D} r_3(dn) + r_3(\perp)) \cdot s_5(n) \cdot Rn$ $\quad + \sum_{d \in D} r_3(d(1-n)) \cdot s_4(d) \cdot s_5(1-n)$
Channels
$K = \sum_{x \in D} r_2(x)(s_3(x) + s_3(\perp)) \cdot K$ $L = \sum_{n=0,1} r_5(n)(s_6(n) + s_6(\perp)) \cdot L$


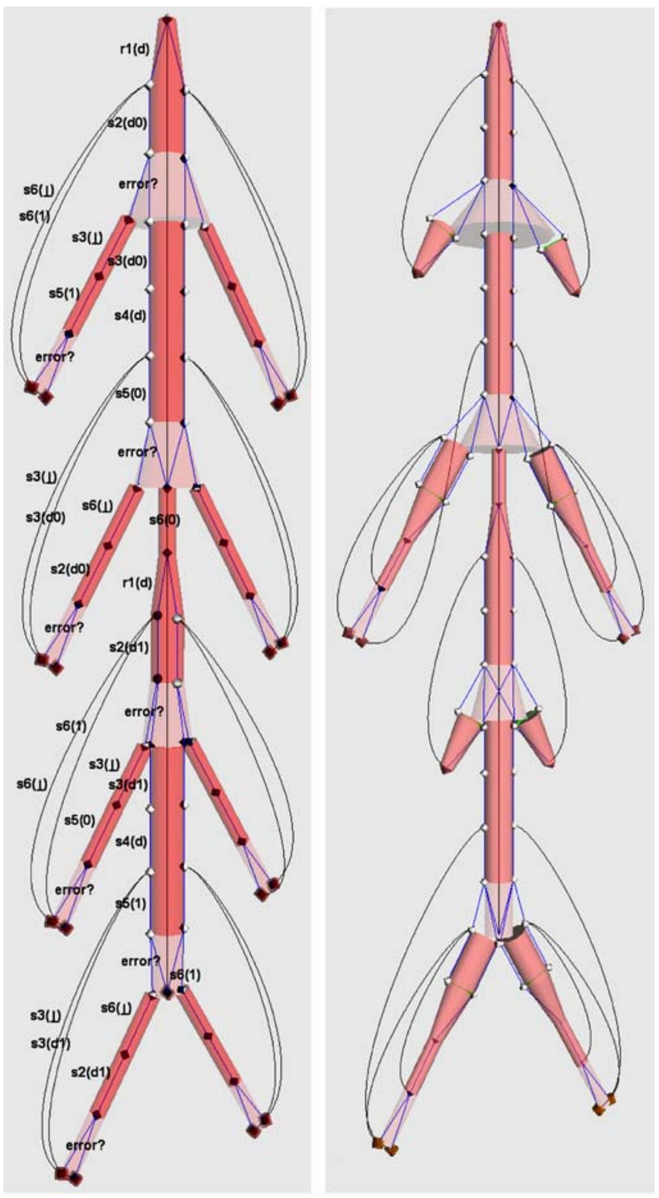


Fig. 7a,b. Communication protocols: ABP reduced modulo strong bisimulation with manual annotations (**a**) and PAR reduced modulo strong bisimulation (**b**)

the behavior for both data elements is identical. This visualization clearly displays regularities in the alternating bit protocol. The behavior in the top half is identical to the behavior in the bottom part, with the distinguishing factor being the control bit sent with the data. The four protrusions on both sides of the visualization represent sections of the protocol where it handles errors in the transmission of the datum (first and third from the top) and errors in the acknowledgement of a transmission (second and fourth from the top). The protocol handles the error and then loops back to the state immediately before the failed transmission to retry.

The PAR protocol globally resembles the ABP but differs in the way errors are handled. Where the ABP assumes nonlossy channels, the communication channels in PAR can lose data. To cope with this additional problem, a timer is integrated into the protocol, which generates a timeout on loss or corruption of a message. This explains why the first and third protrusions are much smaller since PAR deals with errors in transmission more efficiently than ABP: on loss or corruption a timeout is delivered directly to the sender, which can then immediately retransmit. The ABP has to use the acknowledgement channel to communicate this. The handling of errors in acknowledgement transmission (second and fourth protrusion) is very similar, except that PAR has an additional possibility of message loss, accounting for the wider section in the visualization, and that errors arising while resending the data can be dealt with quicker thanks to the timeout (accounting for the shorter loop inside the PAR error handling).

3.3 A modular jack system

This section deals with the communication protocol for a modular jack system. The protocol regulates the communication between an expandable set of industrial jack platforms, allowing the entire set of jacks to be operated from the controls of any one jack. All jacks communicate via a ring-shaped shared bus. The protocol was formally modeled and analyzed in [8], and the exact formulation of the protocol that we visualize is distributed with the μCRL toolset [4]. Figure 8 shows the visualization of the protocol when it has to synchronize two separate jack platforms. One of the immediate features is the existence of two separate but symmetric legs (a). Although this feature is very apparent from the picture, the researchers involved did not realize this until they observed the picture. Generally, the protocol for k jacks has k independent legs.

Another feature that is hard to extract using conventional analysis is the fact that the protocol clearly starts with an initialization phase (Fig. 8e), the end of which is marked by a large number of returning backpointers (see arrows in Fig. 8). Looking at the formal description the initialization phase assigns a consecutive global numbering to all jacks, with each jack having equal opportunity to become the first. This also explains why the two sections are symmetrical; after all, the observable behavior of the entire setup should be insensitive to the numbering scheme used. To illustrate this in Fig. 8, we marked those states in the system in which a jack has been numbered 1 (dark area in the right leg). This implies that in the remaining states this jack has been numbered 0, and that, during the initialization, both jacks are numbered 0 (f).

Another interesting observation is that the general behavior of a system of two platforms is visually very similar to the behavior of a system of three platforms, consisting of approximately 3000 states. Figure 9 shows such a behavior. Notice the three identical sections and how different subsections of the graph visually correspond to

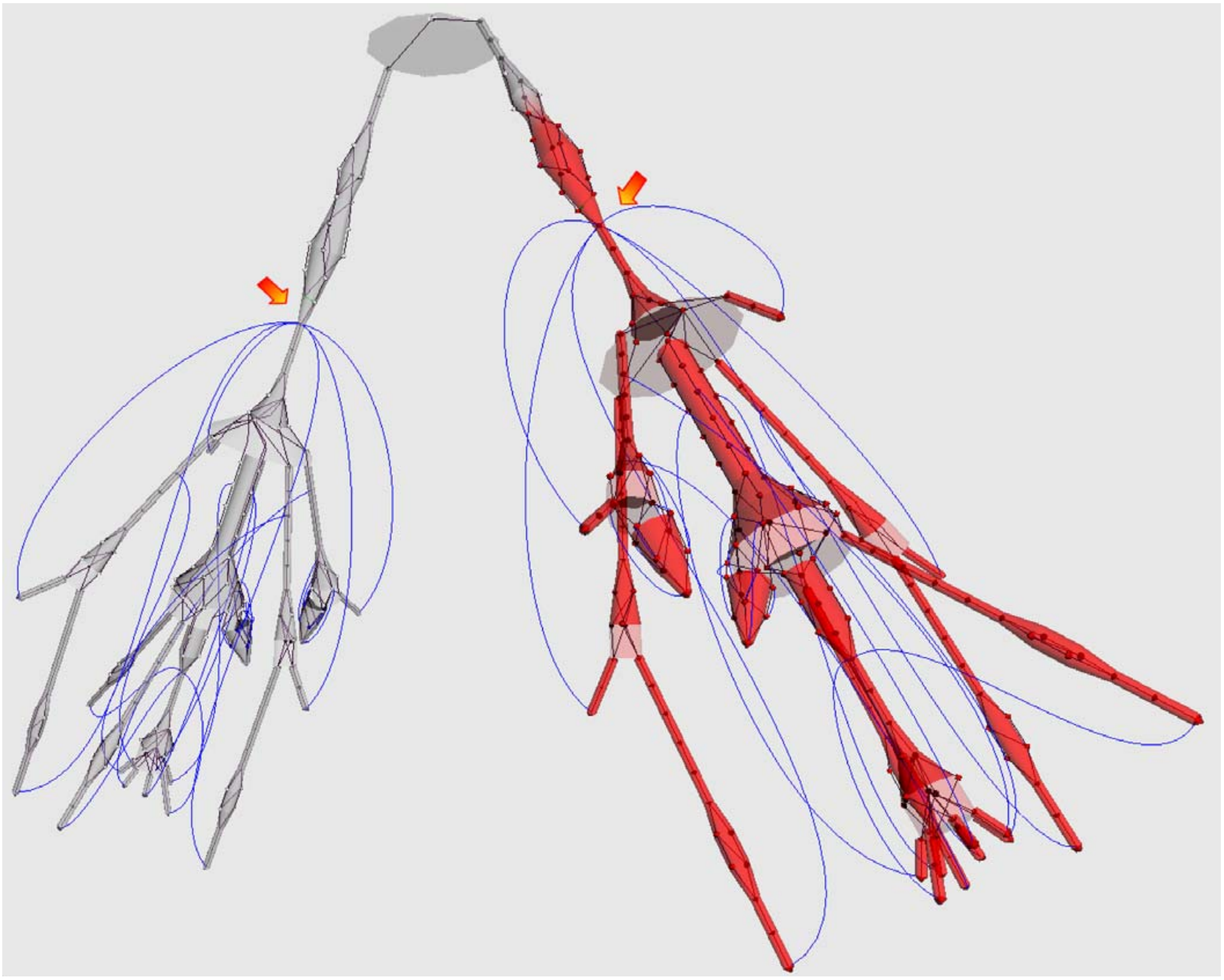


Fig. 8. Modular jack communication protocol for 2 jacks

sections of the two-platform graph. The fact that similar graphs give similar pictures is a major advantage of this method, allowing the application of insight gained into simplified versions of behavior to more complex behaviors. Figure 9 also depicts the full state spaces of five and seven communicating jack platforms, the latter consisting of over one million states. This visualization also shows a bug in the protocol: the small encircled section that splits off during the initialization phase of the protocol has no returning backpointers, which means that states at the end of this appendix are deadlock states (d). This was not the case in the two-jack version. Note that these features can also easily be spotted by conventional analysis (actually, in this case they have been, see [8]); nevertheless a picture in which you can actually point out the bug is a great asset in communication.

We can also use this visualization method to inspect sections of the protocol in detail. Figure 10a shows a section that deals with the bus communication of the three-jack system. When one of the jacks is instructed to perform a common action (i.e., move all jack platforms up synchronously), all jacks are first brought into standby mode. From the top two states (which are actually bisimilar) the first jack broadcasts a “ready for standby” message to all other jacks (1). These messages can be received by the other jacks in any order in which is explained the branching at (2). In the two (bisimilar) top states at (3), both of the other jacks have received the message. The next jack in line broadcasts its “ready for standby” message at (3), and this process repeats itself for every jack in the system. Finally, at (4) the originating jack broadcasts an “all move up” message to the other jacks. In the five- and seven-jack systems in Fig. 9, this typical communication pattern (b) is also abundant, which leads us to estimate that in these graphs at least 50% of the states is related to communication activities.

Another interesting feature is the ability to perform stochastic analysis (c) on different automata. Assuming each transition has an equal probability of occurring, we

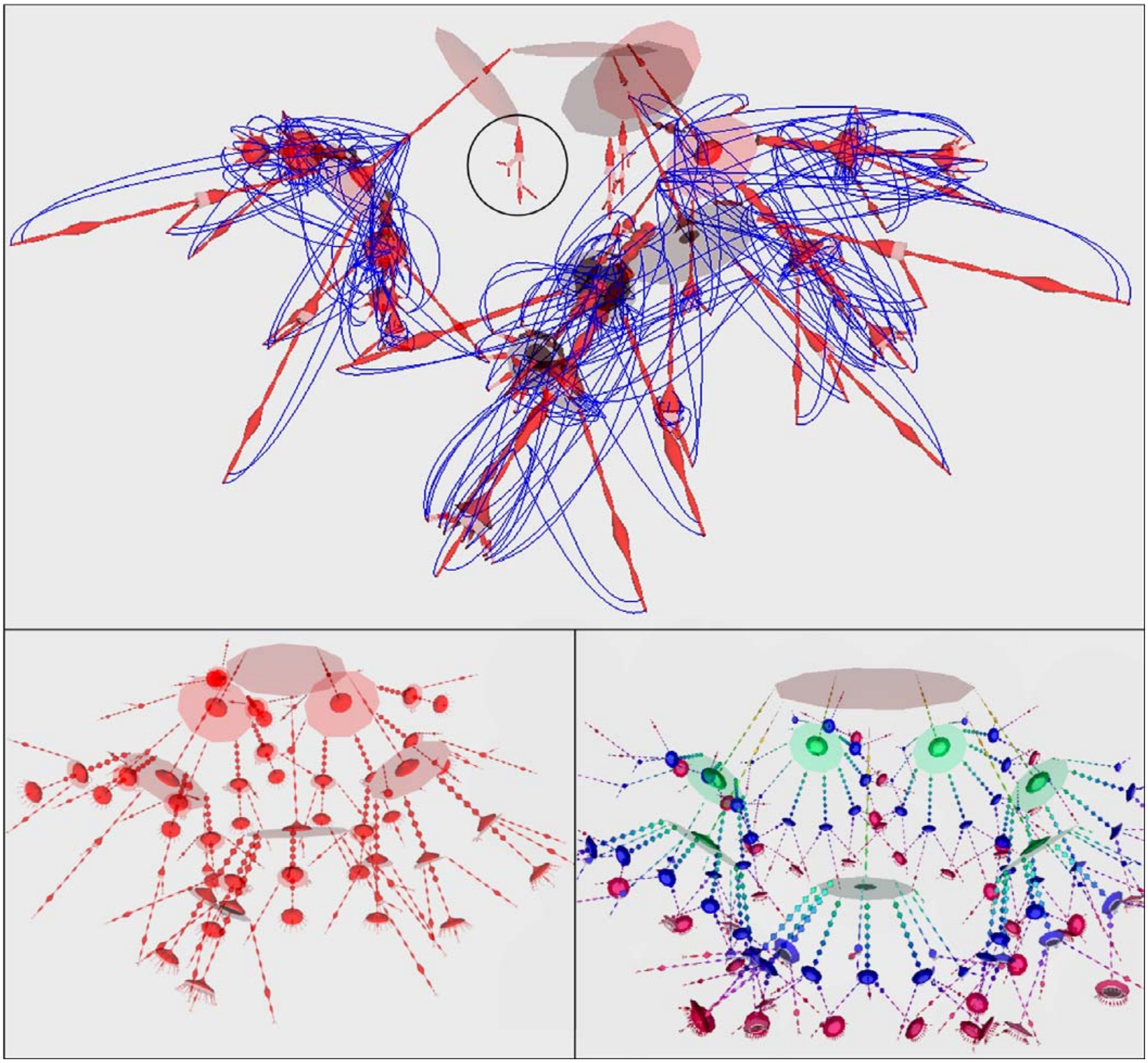


Fig. 9. Protocols for a setup of 3 jacks (2860 states), 5 jacks (70 926 states), and 7 jacks (1 025 844 states), respectively. Backpointers are hidden for the latter two

can gain insight into which states in the process have a relatively high probability of being visited by simulating a random walk. We can then visualize this information by coloring clusters based on these probabilities. Figure 10b shows such a visualization, based on one part of the two-leg-jack protocol. In this case high-probability (dark) sections are located directly behind incoming backpointers and in the two small extrusions on the left of the picture due to the large number of short cycles there. These types of pictures are even more useful if real-world probability data are used. This would make it possible to make a clear distinction between states that are in those parts of the automaton that deal with, for example, error handling and states that are part of the main body of the process.

3.4 The link layer of IEEE 1394

The final example deals with the link layer of the IEEE 1394-1995 [11] (also known as FireWire or I. Link) communication protocol. FireWire is a widely used high-speed serial protocol. Some effort was put into formally analyzing FireWire, resulting in a formal model of the link layer, which provides an interface between the high-level transaction layer and the physical layer [15]. Based on this model, the state transition system of two FireWire nodes sharing a common databus was generated, in which we abstracted from the data being transferred. We ended up with a state space of approximately 25 000 nodes, which is shown in Fig. 11.

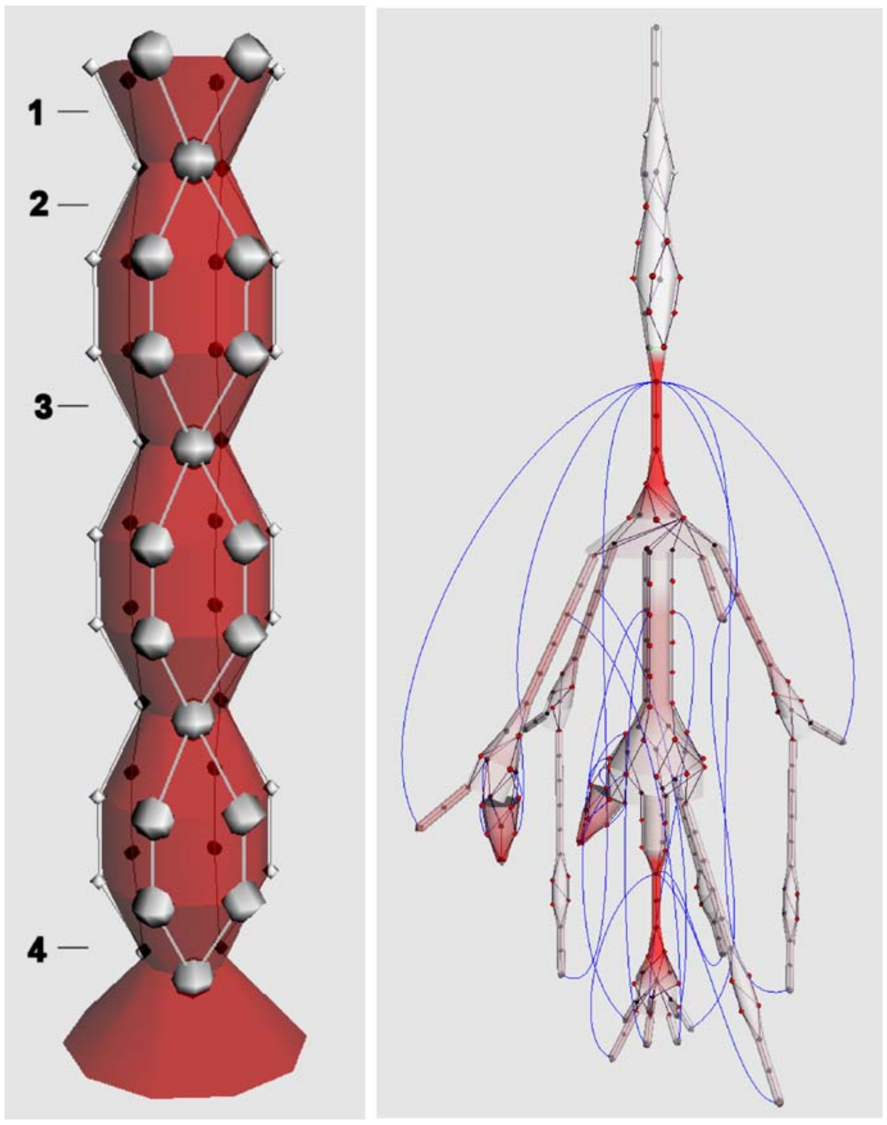


Fig. 10a,b. Detail section of three-jack protocol (**a**) and stochastic analysis of two-jack protocol (**b**)

One of the most notable features of this visualization is the similarity between the two disk-shaped "bulks" at the top and bottom. These are communication phases (b), since the number of states locally expands and then converges again. Both are separated by a relatively thin funnel, consisting of a comparatively small number of states. This type of information is not easily extracted by conventional methods yet can be very useful for testing purposes. During a test run it makes sense to flag states in the funnel to see if these are reached, which indicates the lower bulk is being tested, too. Also, testing in general lends itself well to visualization, since it probably would be desirable to run test traces that cover as much of the state space as possible. A visualization technique such as this one could make it much easier to see what parts of the state space are already covered by tests, for example by rendering a trace in the actual visualization (Fig. 11).

Another question that comes to mind is why we are seeing two similar bulks (Fig. 11a). Although the general behavior for both is almost identical, their state vectors differ. If we could determine a value of a state variable that is unique for the bottom bulk, we could pin a semantic meaning to that specific part of the graph. A naive way of doing this would be to color states based on a specific value for a state variable. If we do this for all possible combinations of variables and values, we can visually determine which variable/value combination might be unique for that part (f). Unfortunately in this case we are dealing with 50 state variables each having a number of possible values. A more efficient way to accomplish this is to select a part of the visualization and subsequently correlating states in this part with the set of states with a particular value for a parameter. The correlation between the two properties x and y over N samples can be given by the

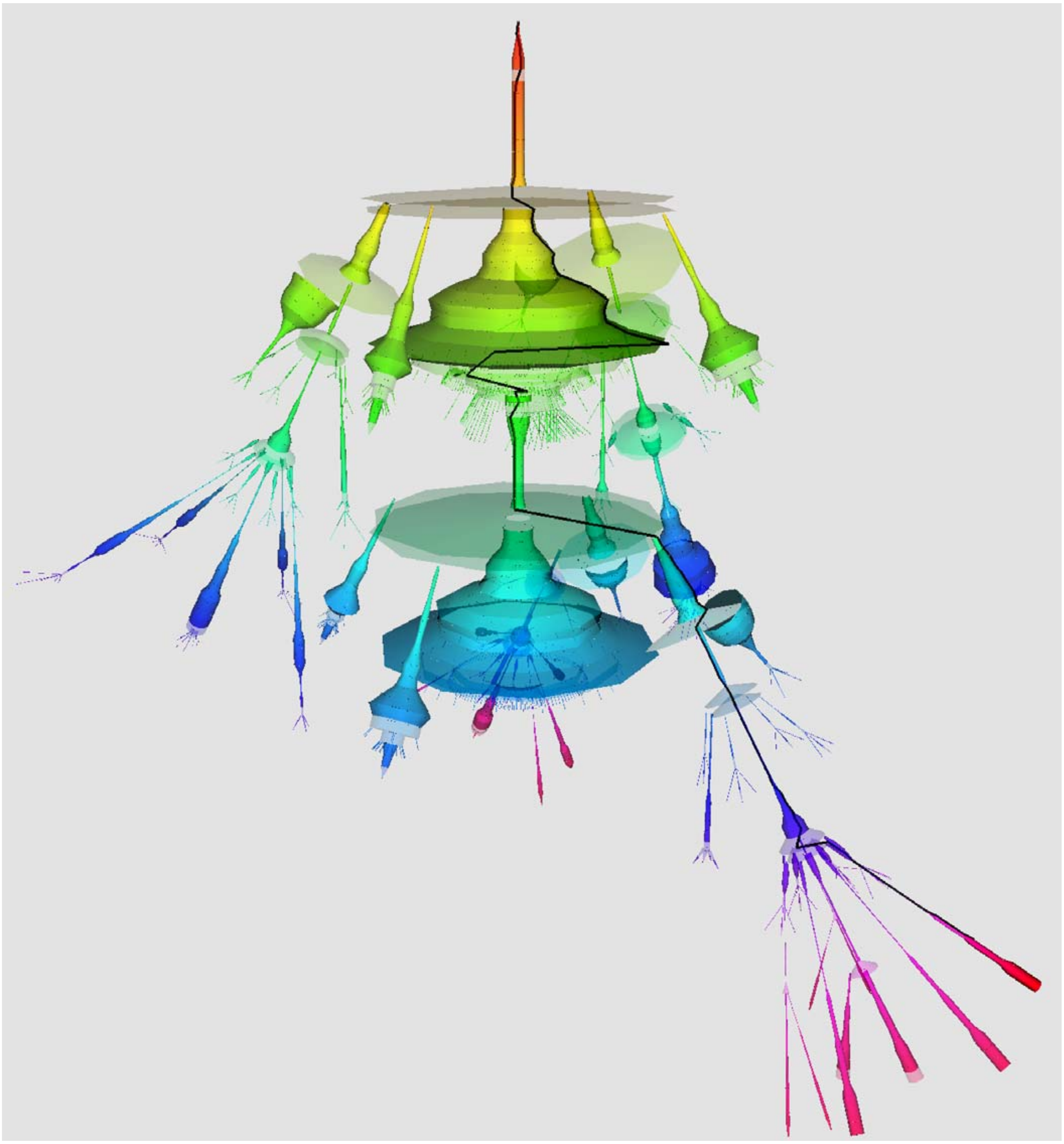


Fig. 11. Rendering a test trace in the visualization of IEEE 1394

Pearson correlation coefficient r:

$$r = \frac{N\sum xy - \sum x \sum y}{\sqrt{(N\sum x^2 - (\sum y)^2)(N\sum y^2 - (\sum y)^2)}} .$$

Suppose we wish to know the correlation between the two properties x and y, informally specified, respectively, as “a node has value v for state parameter p_i” and “a node is an element of a selected set of nodes S.” If we substitute a numeric value of 1 for true and 0 for false, we obtain a binary value pair (x, y) for each of the N nodes. The correlation coefficient can then be computed by substituting:

$$\sum x^2 = \sum x = |n \in V : n.p_i = v| ,$$
$$\sum y^2 = \sum y = |S| ,$$
$$\sum xy = |(n \in V : n.p_i = v) \cap S| .$$

Computing r for all possible combinations of p_i and v yields a list of correlation factors between -1 and 1. A value close to 1 indicates that a property $p_i = v$ is typical for the selected region. A value of -1 indicates that none of the nodes in S has the property $p_i = v$, while all other nodes do. If we do this for all possible combinations of state parameters and values, we end up with a list of correlation factors, of which the ones closest to 1 indicate the best match.

Applying this method on the nodes in the bottom part of Fig. 12 gives us one parameter/value combination that is an almost perfect match. These states are marked in Fig. 12a. Looking back at the formal specification we observed that the parameter in question was a Boolean array b of length two that keeps track of which nodes have already requested use of the bus during a so-called fairness interval. A fairness interval is a fixed amount of time during which all connected nodes should have had the opportunity to use the shared bus if they needed to. All nodes in the marked (darker) set in Fig. 12a have a value of $[\mathit{true}, \mathit{true}]$ for b, indicating that in the bottom bulk both nodes have already requested the bus during the current fairness interval. From this information we can deduce that in the top part a single node has requested use. To illustrate this, Fig. 12b shows the collection of states where $b = [\mathit{false}, \mathit{true}]$.

Although we expected the top bulk to be symmetrical, in fact it is not. The fanlike marked section in Fig. 8b, for example, is not repeated in the top section. There are subtle differences in symmetry, which means that the simulated behavior of the system differs slightly based on which one of the two nodes requests the bus first in the upper bulk of the protocol. Whether this difference lies in the protocol itself or in the formal description of the protocol turns out to be a question that neither we nor the author of [15] has been able to answer yet.

3.5 The VLTS Benchmark set

The Very Large Transition System (VLTS) Benchmark set [7] is a recent set of currently 40 real-world labeled transition systems, meant to serve as a test set for scientific tools and algorithms dealing with transition systems. Sizes of the transition systems in the set vary from a few hundred nodes to 33 million nodes. We applied our visualization algorithm to the transition systems in the set, up to the point where we ran out of physical memory. Figure 13 shows a subset of this collection; the full version can be viewed at `http://www.win.tue.nl/~fvham/fsm/`. Visual representation of these large graphs allows for a number of hypotheses that can be made almost immediately. Firstly, it seems like *vasy_25_25* and *vasy_40_60* are both very regular graphs, although the fact that *vasy_25_25* is a line graph could also be deduced from both the number of nodes and edges and the fact that it is connected. Secondly, *cwi_371_641* bears a strong resemblance to the state space of the FireWire protocol in the previous section and is in fact generated from the same formal description [15]. Such strong global similarities between a graph of 25k and a graph of 371k nodes could not have been found with any conventional tool, or even by comparing these graphs computationally node by node. A closer inspection of the visualizations reveals that *vasy_83_325* and *vasy_166_651* have very similar representations; they also happen to have come from the same industrial application. Apart from these observations, minor observations such as the fact that *vasy_8_38* is symmetrical and has a large number of deadlocks, con-

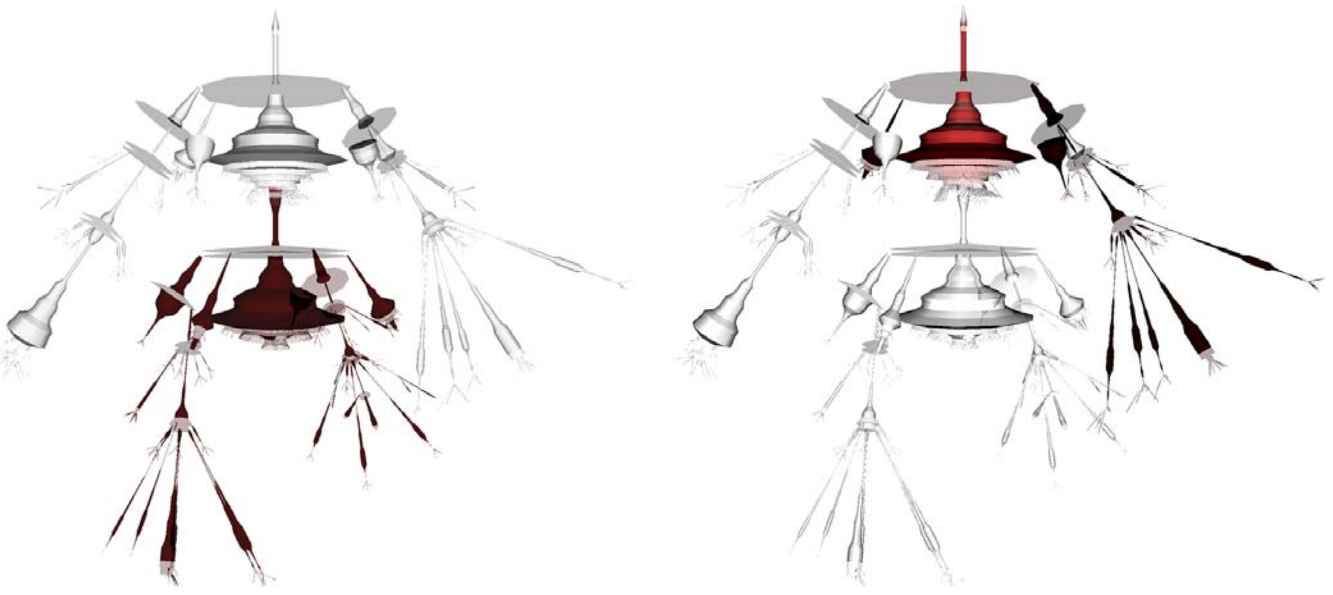


Fig. 12a,b. States where state variable $b = [\mathit{true}, \mathit{true}]$ (**a**) and states where $b = [\mathit{false}, \mathit{true}]$ (**b**)

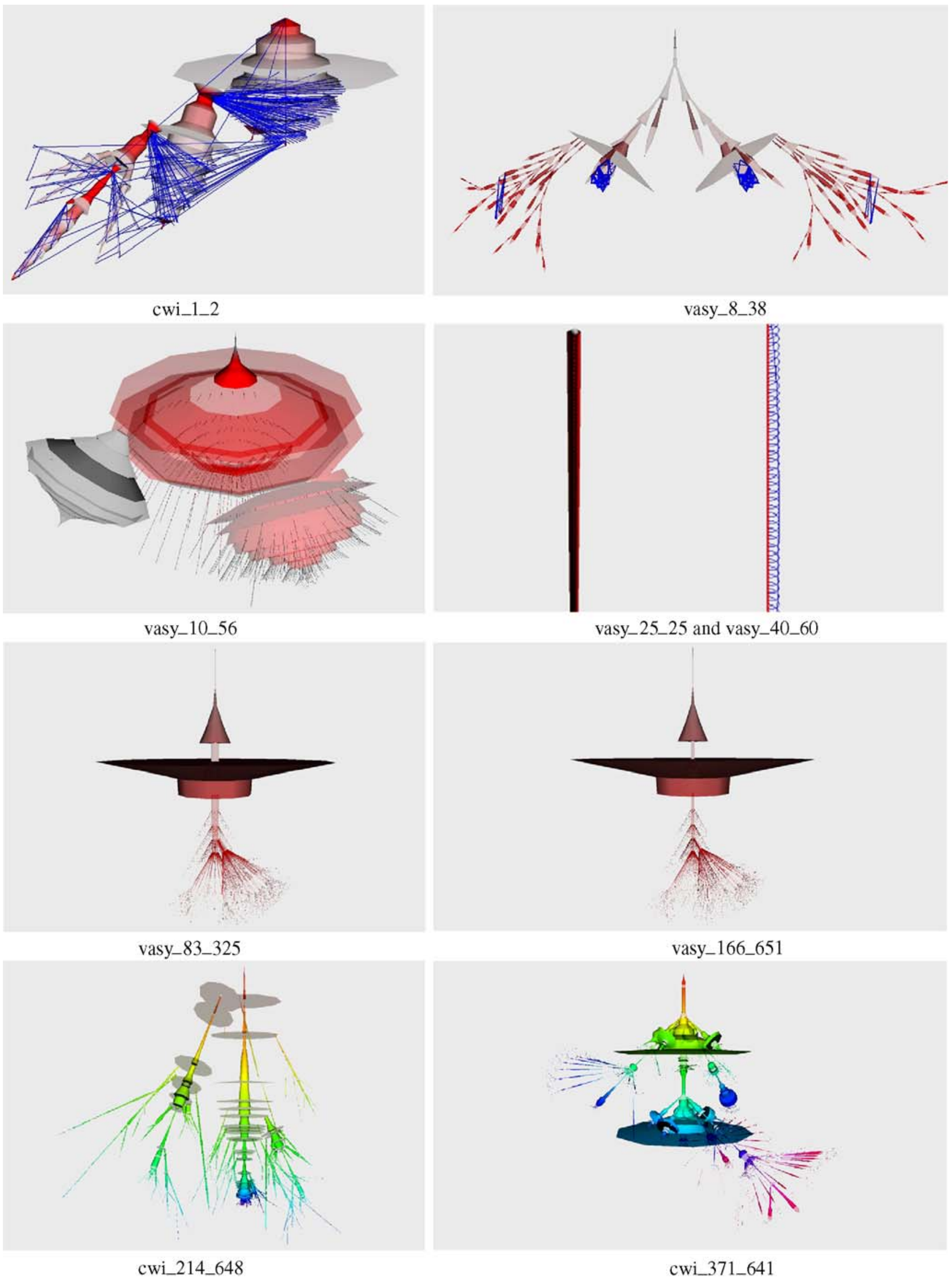


Fig. 13. A number of visualizations of transition systems taken from the VLTS benchmark set

sidering the relatively small number of backpointers, or the fact that *cwi_214_648* has a strong branching structure can prove very valuable when trying to understand the global structure of a state space.

4 Conclusions

In this paper we have made a case for the use of a new visualization technique to gain more understanding of large state spaces. Its main advantages over the few existing visualization techniques are:

– **High scalability**: By not displaying each individual state and transition we can effectively visualize a number of nodes that is at least two orders of magnitude larger than the best conventional techniques. On a medium desktop PC with a high-end graphics card we were able to display visualizations of graphs consisting of over a million nodes. At the same time, the user is still able to interactively zoom in on parts of interest and view individual transitions if needed.
– **Predictability**: In contrast to some other popular graph layout approaches (e.g., force-directed methods or simulated annealing) this method is highly predictable. As a consequence, similar input graphs also lead to similar looking visualizations. This allows us to compare graphs that are generated from similar formal descriptions but have widely differing sizes.
– **Speed**: The performance of the method for large graphs is currently limited by available memory and the speed of the graphics card. The algorithm that generates the global layouts has a time complexity that is linear in both the number of states and the number of edges. See Table 2 for details. The visualization itself is instantaneous.
– **Interactivity**: Since it is impossible to display all information related to a state transition graph in a single picture, interaction is critical. In our application, the user is enabled to zoom into subsections of interest, color parts of the system based on state vector values or transition label values, query state variables, perform stochastic analysis, and much more.

This does not mean, however, that this technique is the ultimate answer to every question one might have for a specific state transition graph. As with any visualization technique it is simply meant as a support tool, to complement other techniques. Nevertheless, a tool that is capable of producing a meaningful image of very large state spaces greatly enhances one's general understanding of these state spaces.

Unfortunately, the method also has its weak points (see also Fig. 14). Firstly, it does not deal well with large graphs that are highly connected, that is, graphs in which the average shortest path length between states is small. In this case, nodes tend to group together in a very small number of clusters, which gives no information on the internal structure of the graph. Secondly, the method is not stable in the sense that in worst-case scenarios, adding a single state to the graph might radically change the global layout if this state connects two previously unconnected sections. Thirdly, the layout algorithm used to position individual nodes in a cluster does not always produce an optimal layout. Finally, we found that researchers whom confronted with visualization of their own state spaces generally had a hard time interpreting the pictures because in most cases they had no reference as to what the state space they were analyzing should look like. The option to mark sections of the visualization based on a specific state parameter or transition label proved a great help in relating the original specification to the shape of the visualization.

Future work on this method will focus on integrating state reduction techniques such as abstraction and bisimulation into the visualization. One can think of the possibility of interactively collapsing similar looking subsections or a visualization of a large state space morphing into a smaller version. We also plan to look into recursively applying this method to more complex sections of graphs, which may split large clusters into a number of smaller ones. Finally, the method currently has a rather large memory footprint, and we are working to improve this.

Table 2. Timing and memory results on a number of real-world graphs, measured on a 2.4-GHz Pentium IV

Nodes	Edges	Time (s)	Memory (MB)
191	244	0.016	0.2
429	592	0.016	0.4
2860	4485	0.047	2
15 409	27 152	0.203	10
18 746	73 043	0.29	16
70 926	145 915	1.187	51
157 604	297 000	11.31	111
164 865	1 619 204	26.34	236
166 464	651 168	13.18	125
214 202	684 419	3.19	167
386 496	1 171 872	16.04	310
1 025 844	2 932 909	93.68	750

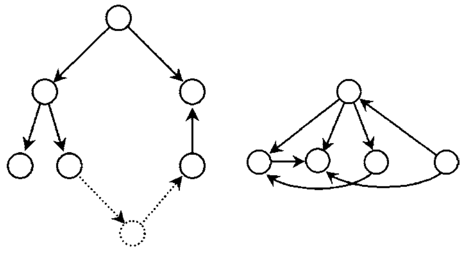


Fig. 14a,b. Situations in which the visualization does not perform well: (**a**) adding a new state to two previously unconnected states and (**b**) graphs with a small average path length

Summarizing, we have not seen any technique that even remotely resembles ours, and we sincerely believe that this visualization technique is a great step forward in understanding the global structure of state transition diagrams. Actually, by looking at and interacting with the visualization tool we became aware of many properties of state spaces that, although sometimes obvious and sometimes more obscure, we had not realized until we saw them.

Acknowledgements. Thanks to Judi Romijn, Huub van de Wetering, and Jack van Wijk for their valuable discussions, comments, and programming work and the Netherlands Organisation for Scientific Research (NWO) for their financial support under Grant 612.000.101.

References

1. Arnold A (1994) Finite transition systems. Prentice Hall, Englewood Cliffs, NJ
2. Baeten J, Weijland P (1991) Process algebra. In: Cambridge tracts in theoretical computer science, vol 18. Cambridge University Press, Cambridge, UK
3. Fernandez J-C, Garavel H, Kerbrat A, Mateescu R, Mounier L, Sighireanu M (1996) CADP – A Protocol Validation and Verification Toolbox. In: Proceedings of the 8th International Conference on Computer Aided Verification. LCNS, vol 1102, Springer, pp 437–440
4. Blom SCC, Fokkink WJ, Groote JF, van Langevelde IA, Lisser B, van de Pol JC (2001) mCRL: A toolset for analysing algebraic specifications. In: Computer-Aided Verification (CAV 2001). Lecture notes in computer science, vol 2102. Springer, Berlin Heidelberg New York, pp 250–254
5. Carrière J, Kazman R (1995) Research report: interacting with huge hierarchies: beyond cone trees. In: Proceedings of the IEEE conference on information visualization. IEEE Press, New York, pp 74–81
6. Gansner ER, North SC (2000) An open graph visualization system and its applications to software engineering. Softw Practice Exper 30(11):1203–1233
7. Garavel H et al. (2003) The VLTS (Very Large Transition System) Benchmark suite. `http://www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html`
8. Groote JF, Pang J, Wouters AG (2003) Analysis of a distributed system for lifting trucks. The Journal of Logic and Algebraic Programming 55(1–2):21–56
9. Groote JF, van Ham F (2003) Large state space visualization. In: Proceedings of TACAS 2003, pp 585–590
10. van Ham F, van de Wetering H, van Wijk JJ (2002) Visualization of state transition graphs. IEEE Trans Visual Comput Graph 8(4):319–329
11. IEEE Computer Society (1996) IEEE standard for a high performance serial bus, Std 1394-1995, August 1996
12. Jéron T, Jard C (1994) 3D layout of reachability graphs of communicating processes. In: Proceedings of the DIMACS international workshop on graph drawing. Springer, Berlin Heidelberg New York, pp 25–32
13. Koren Y, Carmel L, Harel D (2001) ACE: A fast multiscale eigenvectors computation for drawing huge graphs. Technical Report MCS01-17, Weizmann Institute Of Science. `http://www.wisdom.weizmann.ac.il/reports.html`
14. van Langevelde IA (2001) A compact file format for labeled transition systems. Technical report SEN-R0102, CWI, Amsterdam, The Netherlands
15. Luttik SP (1997) Description and formal specification of the link layer of P1394. Technical report SEN-R9706, CWI, Amsterdam, The Netherlands
16. Mauw S, Veltink GJ (eds) (1993) Algebraic specifications of communication protocols. Cambridge tracts in theoretical computer science, vol 36. Cambridge University Press, Cambridge, UK
17. Robertson GG, Mackinlay JD, Card SK (1991) Cone trees: animated 3D visualizations of hierarchical information. In: Proceedings of the conference on human factors in computing systems (CHI '91), pp 189–194
18. Tutte W (1963) How to draw a graph. Proc Lond Math Soc 3(13):743–768