MOVIS: A system for Visualizing Distributed Mobile Object Environments

Yaniv Frishman*and Ayellet Tal†

Technion - Israel Institute of Technology

Abstract

This paper presents MOVIS – a system for visualizing mobile object frameworks. In such frameworks, the objects can migrate to remote hosts, along with their state and behavior, while the application is running. An innovative graph–based visualization is used to depict the physical and the logical connections in the distributed object network. Scalability is achieved by using a focus+context technique jointly with a user-steered clustering algorithm. In addition, an event synchronization model for mobile objects is presented. The system has been applied to visualizing several mobile object applications.

Index Terms – Distributed software visualization, mobile objects, dynamic graph layout

1 Introduction

In recent years, distributed objects have become prominent in the design of distributed applications [41]. Mobile objects are a natural evolution of the distributed objects concept [1,22,35,40,42]. The mobile object paradigm allows programs to migrate to remote hosts while they are running. It

*e-mail: frishman@tx.technion.ac.il

†e-mail:ayellet@ee.technion.ac.il

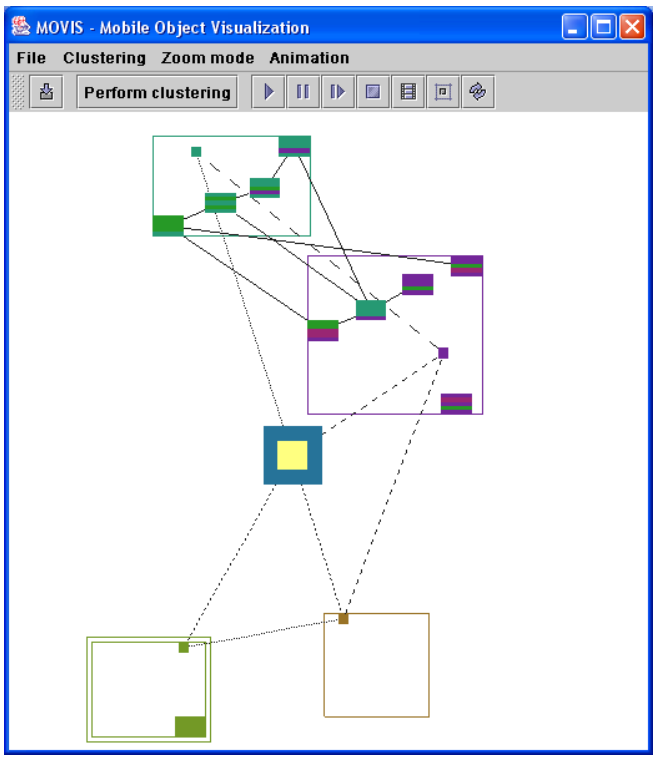


Figure 1: MOVIS user interface. Small rectangles represent mobile objects. Color stripes show their movement history. Big rectangles represent the cores the objects reside in. Dashed lines represent physical communication between cores. Higher communication frequency is indicated by a higher frequency of alternation in the lines. Solid lines represent logical connections between objects. The square in the middle of the figure represents several cores which have been collapsed. The rectangle with a double boundary was selected by the user as the current focus of attention core.

offers scalability, availability and flexibility advantages compared to other methods of creating distributed applications. However, such systems are more difficult to design and debug, two tasks in which visualization can greatly assist. This paper addresses the challenging problem of visualizing mobile objects.

Mobile objects have two distinctive features. The first feature is *code mobility*: objects can migrate to remote hosts, together with their state and behavior, while the application is running. We refer to the processes hosting mobile objects as *cores*. The second feature of mobile objects is *location transparency*, which allows the programmer to make calls to objects regardless of their

current location. Since the location of objects may change over time, provisions must be supplied in order to track referenced objects. Unlike regular distributed objects, in which the location of a remote object is fixed, when making a call using a reference to a mobile object, the parameters may pass through several intermediary cores until reaching the called object. The introduction of intermediary cores allows for a more scalable, lazy update of the location of a referenced object [22].

Although research on mobile objects is widespread, visualization of such frameworks has hardly been done. As far as we know, the only work in this field includes [51, 53]. These systems fall short in several aspects, including the types of events generated and visualized and visualization consistency and scalability, which are addressed in this paper.

This paper makes the following contributions: First, we present MOVIS (Mobile Object Visualization), a system for visualization of distributed mobile object environments. Second, we discuss the requirements of a visualization system for mobile objects. Third, a graph-based visualization that concurrently shows the physical connections in the computer network as well as the logical relations between the mobile objects is presented (see Figure 1). Fourth, a context-sensitive focus+context fisheye type display technique is suggested in order to provide hierarchical information display and support scalability. Fifth, a clustering algorithm, which is affected by nodes of interest to the user, is presented. Sixth, we present a model for event synchronization that is used to guarantee visualization consistency. Finally, we propose a method in which events are automatically generated, avoiding additional work by the programmer of the application.

The rest of this paper is structured as follows: Section 2 discusses related work. In Section 3, the requirements of a mobile object visualization system are discussed. Our visualization is presented in Section 4. Section 5 addresses consistency. Section 6 discusses scalability. Section 7 discusses our graph drawing algorithm. Section 8 discusses implementation issues. Results are presented in Section 9. Finally, Section 10 concludes and discusses future directions.

Preliminary versions of this research have been presented in [14, 15].

2 Related Work

This paper falls at the crossroads between two research fields: distributed software visualization and graph drawing.

Distributed software visualization: Several tools have been developed for visualizing parallel and distributed programs [30]. The PVanim system [48] is a toolkit for creating visualizations of the execution of PVM programs. PARADE [45] in an environment for developing visualizations of parallel and distributed programs. In [37], tracing of CORBA [41] remote procedure calls is used to analyze runtime activities and look for anomalous behavior. Vade [38] is a distributed algorithm animation system in which visualizations can be created and executed on a web page on the client's machine. Pablo [44] provides analysis and presentation of performance data for massively parallel distributed memory systems. Jinsight [43] is a system for the visual exploration of the run-time behavior of complex Java programs.

Although research on mobile objects is widespread, visualization of such frameworks has hardly been done. In [51], a modification of the process-time diagram, adopted from XPVM [27], is used as the means of visualization. The creation, destruction and movement of mobile objects are visualized. Event synchronization is handled by timestamps and ordering rules. This system has a few drawbacks. It requires manual annotation of source code in order to generate events. The system does not visualize the physical connections between machines nor does it display the logical connections between objects. Finally, it is not scalable. In [53] a visualization tool used to debug mobile objects is presented. This tool is concerned mainly with checking the mobility of objects as a function of time and identifying movement hotspots. Visualizations offered include an object location display and movement history for an object. The system does not visualize communication between objects or between computers hosting the objects. Visualization consistency and scalability to large numbers of objects in not addressed. In this paper we discuss a different approach to the visualization of mobile object frameworks, attempting to solve these problems.

Graph drawing: The general problem of drawing graphs, e.g., assigning coordinates to graph vertices, edges and other elements, has been extensively studied [9, 26, 46]. One popular technique

is force-directed layout, which uses physical analogies in order to converge to an aesthetically pleasing drawing [25,26,46]. Drawing non-point vertices using this approach is discussed in [7,17, 21]. Extending force-directed algorithms for drawing large graphs is discussed in [19, 20, 28, 49].

Work on clustered graph drawing is less widespread. In [50], a divide and conquer approach, in which each cluster is laid out separately and then the clusters are composed to form the graph, is used. In [11], a method of drawing the clustering hierarchies of the graph using different Z coordinates in a 3D view is discussed. See also [3, 26] for a discussion of clustered and compound graph layout.

Many applications require *dynamic graph drawing*, i.e., the ability of modifying the graph [10, 26, 39]. Different types of graph modifications may be performed. These include adding or removing vertices and edges. The challenge in dynamic graph drawing is to compute a new layout that is both aesthetically pleasing as it stands and fits well into the sequence of drawings of the evolving graph. The latter criterion has been termed *preserving the mental map* [36] or *dynamic stability* [39].

Several algorithms address the problem of offline dynamic graph drawing, where the entire sequence is known in advance. In [10], a meta-graph built using information from the entire graph sequence, is used in order to maintain the mental map. In [31] a stratified, abstracted version of the graph is used to expose its underlying structure. An offline force directed algorithm is used in [13] in order to create 2D and 3D animations of evolving graphs. Creating smooth animation between changing sequences of graphs is addressed in [4].

An online graph drawing algorithm is discussed in [33], where a cost function that takes both aesthetic and stability considerations into account, is defined and used. Unfortunately, computing this function is very expensive (45 seconds for a 63 node graph). Drawing constrained graphs has also been addressed. Incremental drawing of DAGs is discussed in [39]. Dynamic drawing of orthogonal and hierarchical graphs is discussed in [18].

The DA-TU system described in [23] allows navigating and interactively clustering huge graphs. In [34] an algorithm that tries to improve the distribution of nodes in a graph while maintaining the mental map is described. Finally, some commercial graph layout packages such as [47,52] contain

provisions for dynamic layout of graphs. As far as we know, none of the above was designed to handle incremental drawing of clustered graphs. Here, we wish to support adding and removing nodes, clusters and edges and moving nodes between clusters.

3 Requirements

This section discusses the requirements of a visualization system for mobile objects:

1. Physical and logical visualization: A mobile object application has two distinct, yet related facets. The first is the physical computer network with the interconnections between the cores. The second is the logical network of mobile objects that can be used to show the connections and interactions between objects. The visualization should display both of these facets. This is important in order to easily detect cases where closely interacting system components are placed on distant nodes. Using the visualization the system architect will detect this inefficiency and modify the logic and layout of the application in order to place such objects close together.

2. Interesting events: In any visualization system, the events that need to be visualized greatly affect the design of the system. In the case of mobile objects, the following interesting events should be visualized:

- Object Movement: The movement of objects between cores while the application is running is the main difference between mobile object frameworks and regular distributed applications. Therefore, a clear and concise depiction of such activities is of great importance.

- Construction/destruction: Being a dynamic, distributed application, both objects and cores may be added or removed during the execution of the application.

- Communication: Being distributed in nature, the messages sent between the different parts of the system play a paramount role during execution of the application and therefore provisions to visualize them should be supplied.

Event generation should be transparent both to the programmer and to the user of the application. Moreover, care must be taken in order to reduce the perturbation of the application caused by

generating the events.

Being able to visualize movement allows easily identifying cases where objects migrate too often, which is inefficient. Visualizing communication can help expose closely interacting objects, which should be placed in close proximity

3. Consistent depiction: In a distributed, asynchronous environment there is no global clock that can be used to synchronize events. This may lead to inconsistent visualizations in which, for example, a message is shown to be received before it is sent. One of the challenges in visualizing distributed systems is creating an animation that provides a consistent depiction of events. This is especially challenging for mobile objects, since parts of the application change their physical location during execution.

4. Scalability: One of the main challenges in software visualization is building a scalable visualization. This is especially important when dealing with networks of computers, which can potentially generate massive amounts of information. A visualization system should be able to process large amounts of data. This should be done while avoiding swamping the user with unnecessary information and without slowing the response of the visualization system to a point where it is no longer useful.

The user should be able to steer the visualization system to display relevant and interesting data out of the large amount of information collected. This control should be interactive, allowing the user to feed back to the system new requests based on the knowledge accumulated while viewing the unfolding visualization. This will allow the user to easily study interesting parts of a large distributed system, for example ones which require tuning.

5. Dynamic graph layout: A graph is a natural way to represent the structure of a software system. In the context of mobile objects, a dynamic, clustered graph is used, as explained in Section 4. Since the graph is dynamic, it is important to produce stable layouts that help maintain the users mental map of the system. This is required in order to avoid distracting the user with confusing changes to the way the graph looks each time it changes.

In the following sections we describe how MOVIS addresses there requirements.

4 Physical and Logical Visualization

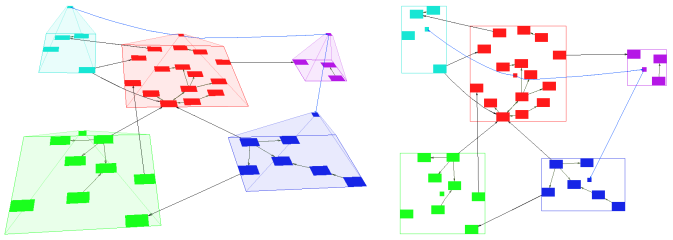
As discussed in Section 3, two simultaneous networks are of interest: the physical network of cores (machines) and the logical relations and interactions between mobile objects. A graph is a natural choice for visualizing a distributed network. In our case, we need to simultaneously visualize two graphs. We use the following definition:

Definition 4.1 ***Clustered Graph:*** *A clustered graph is an ordered quadruple $G = (V, C, E_v, E_c)$, where V is the node set, C is a set of clusters which form a partition of the node set V, E_v is the set of edges between nodes $E_v \subseteq \{ (v_i, v_j) | i \neq j, v_i, v_j \in V\}$ and E_c is the set of cluster-cluster edges $E_c \subseteq \{(C_i, C_j) | i \neq j, C_i, C_j \in C\}$.*

A clustered graph is a natural choice for displaying the simultaneous physical and logical graphs, as demonstrated in Figure 1. Every mobile object is depicted by a node in the graph. The logical connections between objects are shown using solid edges connecting the nodes. In order to overlay the physical structure of the network, clusters are used. Each core is represented by a cluster that contains all of the objects currently residing in that core. Dashed cluster-cluster edges are used to represent physical connections between cores (see Figure 1), as opposed to logical relations that exist between objects.

The graph can be displayed in three dimensions, as illustrated in Figure 2 (a). Edges between nodes, showing relations between mobile objects, are drawn on the lower plane, while cluster-cluster edges, showing physical connections between cores, are drawn on the upper plane. In 3D, a cluster is drawn as a semi-transparent pyramid. A small dummy node is added to each cluster, drawn at the apex of the pyramid and serves as the endpoint of cluster-cluster edges. One of our guidelines in creating this visualization is being able to collapse the 3D view into a 2D view in a natural and comprehensible way, as illustrated in Figure 2 (b), which shows a 2D drawing of the graph from Figure 2 (a).

We use several techniques and attributes in order to display information in this graph. Each cluster boundary is drawn using a different color. This helps the user track the different clusters while changes are performed to the graph during the visualization.



(a) 3D view (b) 2D view

Figure 2: Visualization of mobile objects in 3D & 2D

Each node is drawn using color strips, as shown in Figure 1. The strips are colored according to the location history of the object. The bottom strip is the current location (e.g. colored with the same color as the cluster the node currently resides in) the strip above corresponds to the previous location, etc. This ”growing stacks” metaphor is similar to the *growing squares* in [12].

In order to create a more scalable and meaningful display, we employ lazy construction of edges. Instead of cluttering the graph with node-node edges showing all of the references between objects, an edge is drawn between two objects once a method call between the objects is detected.

In addition to the existence of communication between objects or cores, the frequency of this communication is of interest to the user. Line patterns are used to convey this information. The higher the frequency of alternation in the dashed lines, the higher the frequency of communication. See for example Figure 1. The sum of two weighted averages is used to calculate the amount of communication between cores. The first is the average number of objects moving between the cores connected by the edge. The second is the average number of remote invocations performed between the two cores. The averages are calculated using a weighted sliding window, taking the last N samples into account.

Some mobile object frameworks [22] allow tagging of specific objects as stable, i.e. objects that remain at the same location throughout their lifetime. This distinction between stable and movable objects is visualized by laying out the objects in each cluster using two concentric circles. The inner circle contains the stable objects while the outer one contains movable ones.

Animation is used in order to show different events. When a new graph layout is performed, for example after an object moves between cores, the positions of nodes, edges and clusters are

linearly interpolated between the old and the new locations. A method call between two remote objects is animated using a lightning bolt icon that moves from the caller to the called object.

5 Visualization Consistency

One of the main challenges in visualizing distributed environments is the accurate depiction of events. Since in asynchronous distributed systems there is no way of knowing the real ordering of events, it is necessary to generate a visualization that is *consistent* with the events [32].

We base our solution to event synchronization on [38], where consistency of distributed environments with static objects was addressed, and extend it to support mobile object frameworks. In [38], the following is assumed:

1. There is a fixed (known) number of processes.
2. A process can perform two types of actions: sending a message to a different process and an internal computation, possibly modifying the process's local state. Receiving a message is considered an internal action.
3. The communication network and processes are reliable.
4. Messages sent by a single process to another process arrive in the order they were sent.
5. The network is asynchronous - there is no universal clock.

Since the visualization process is part of the distributed environment, it cannot know the relative order of actions performed by different processes. A way to solve this difficulty is to introduce *semantic causality*.

Definition 5.1 *With respect to a given algorithm run r, we say that an event e in r* semantically causes *e', denoted by $e \rightarrow e'$, if one of the following holds:*

1. *e and e' are on the same process, e occurs before e' and the user specified that they are semantically dependant.*

2. *e and e' are on two different processes connected by a communication channel, e is a* send *event and e' is the corresponding* receive *event.*

3. *There is an event e'' such that $e \rightarrow e''$ and $e'' \rightarrow e'$.*

Let e and e' be two events of the algorithm. Let $An(e)$ and $An(e')$ be the animation segments of these events, respectively. We say that an animation $An(e)$ precedes an animation $An(e')$, denoted by $An(e) \prec An(e')$, if $An(e)$ completes before $An(e')$ starts.

The following theorem has been proved in [38]:

Theorem 5.1 *An animation is consistent with the execution of the algorithm if and only if for every two algorithm events e and e', such that $e \rightarrow e'$ also $An(e) \prec An(e')$.*

That is, in order to ensure that the animation is consistent with the execution of the algorithm, we have to ensure that for every two events e and e', if $e \rightarrow e'$ then $An(e) \prec An(e')$.

A possible implementation of this requirement is called *receive synchronization*. In this method, reports of send and receive events are sent to the animation system immediately after they take place and there is no delay in the execution of the algorithm. The animation of the receive event is delayed until the corresponding send event has been animated.

We now turn our attention to mobile object environments. The main differences between this model and the distributed environments model, in the context of consistency, are:

1. Assumption 1 is violated. Both cores and objects might join or leave the network.

2. Objects might move between cores.

3. Assumption 4 is violated. Since objects might move, messages sent by a single object to another might be received out of order.

The first problem is addressed as follows. Dynamic creation and deletion of cores and objects are modeled as internal messages. A core / object is introduced to the animation system after its internal create event is received. A core / object is deleted from the animation system once a *deletion event* is received and all proceeding events have been animated.

To solve the second problem, object movement between locations is modeled as a method call between the sending and receiving cores. The parameters passed include the state and behavior (code) of the object that is being moved from one core to the other. This approach allows us to synchronize the events emitted by objects, even when they migrate to remote cores. It is ensured that the movement event will be animated before any events generated at the destination core are shown. Hence, when treating object movement as a method call between cores, we are able to revert to using existing event synchronization algorithms.

The third problem, out-of-order messages, should be solved by the middleware or the application. It is not a visualization problem, but rather an inherent problem. When this is solved, all that remains is to solve possible out-of-order reception of messages by the visualization system. This can be done by adding an event counter to each object and using the *receive synchronization* technique described above for visualization.

We have chosen to perform synchronization at the core level. Using a finer-grained approach requires extensive profiling of the application which may slow down execution considerably. Much like regular distributed applications, each core is viewed as a separate process, and events notifying about communication between separate cores and activities internal to each core are emitted and synchronized. The internal events in each core are serialized. This may add redundant dependencies between activities that are independent in a core but is guaranteed to create a consistent visualization. The alternative of asking the user to explicitly define dependencies is not viable in the context of our problem.

Messages sent between cores are modeled as messages sent between processes. The dependency between receiving the parameters for a message call and forwarding the parameters to the next core on the way to the destination core is handled automatically since these are two events that occur at the same core, one after the other. This is also true for messages sending the return value back to the caller core.

Events showing average information that is periodically updated are not synchronized. For example, in our system events notifying the amount of communication between cores are periodically generated, yet not synchronized.

6 Visualization Scalability

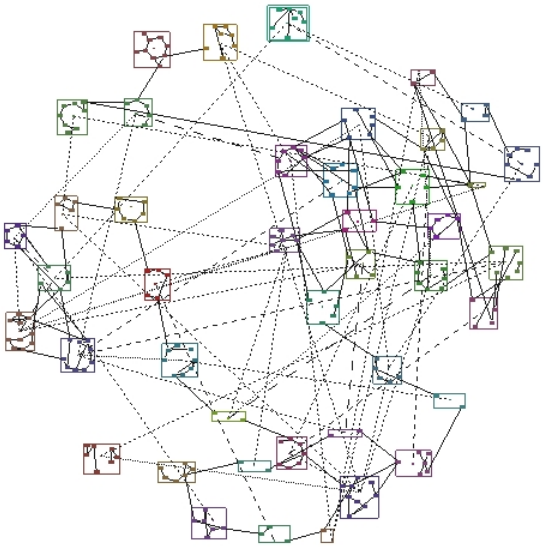
As the number of objects and cores increases, the visualization might get cluttered with information. Gaining any insight from the visualization will become increasingly difficult. In this section we present a context sensitive focus + context technique that alleviates this problem.

6.1 Levels of Detail

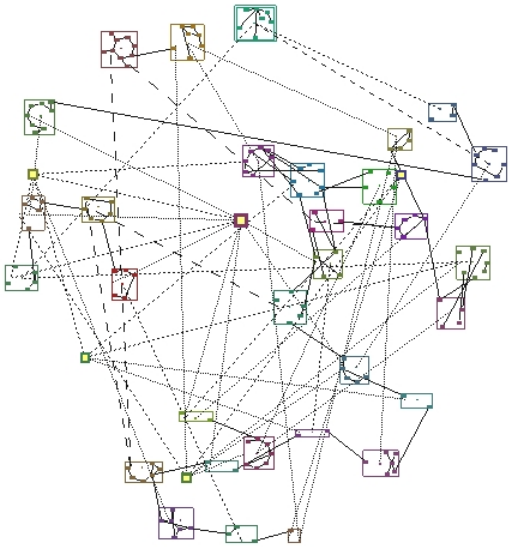
The visualization should provide the user with an overview of the graph while at the same time allowing focusing on specific, user-defined areas in order to get more detailed information [6, 16]. To achieve these goals, a hierarchy of levels of detail is defined, allowing different parts of the graph to be displayed in different levels of detail.

At the highest level, full information is displayed, as shown in Figure 3(a). The next level of the hierarchy omits information about the objects residing in each core and the logical connections between objects. Instead of displaying a cluster for each core in the network, a single node is used to depict each core. As before, cluster–cluster edges are used to convey the physical connections to other cores in the network, as demonstrated in Figure 3(b). The final level combines several cores into one node in the display. This allows collapsing un-interesting parts of the graph into a small display area while still showing the user the overall structure of the graph. The size of such nodes is proportional to the number of cores they depict. Figures 3(c) and (d) demonstrate graphs containing nodes of various levels of detail. Note the stability of the layouts and the way nodes are collapsed as the level of detail is decreased.

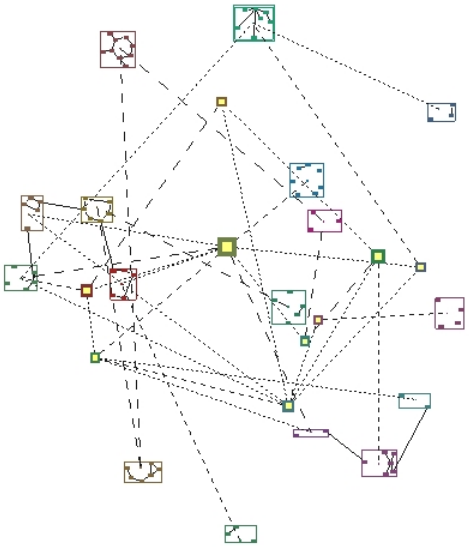
The user has several methods to control which parts of the graph will be displayed in which level of detail. The first is selecting focal nodes (cores) that are of primary interest to the user and thus should be displayed with full detail. The second method is navigating the graph using zoom-in and zoom-out operations. The third is choosing the total number of nodes to be displayed in the graph and letting the system cluster the graph nodes accordingly. Once the user selects focus nodes, a clustering algorithm is employed in order to decide at what level of detail each core will be displayed, as described in the next subsection.



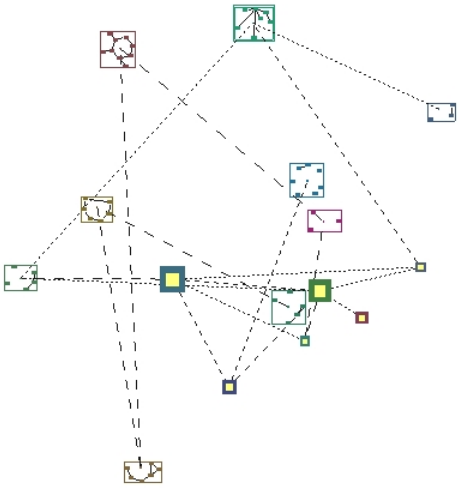
(a) Original graph – 43 clusters



(b) Clustering to 36 clusters



(c) Clustering to 25 clusters



(d) Clustering to 15 clusters

Figure 3: Levels of detail. Several visualizations of the same mobile object network are shown. Parts of the graph are progressively collapsed. Note the stability in the layouts and the conservation of the overall structure of the graph.

Zoom–in and zoom–out operations are animated smoothly. The old nodes fade out of the graph while the new nodes fade in. Next, the new nodes smoothly move to their final location. This helps the user understand the changes to the graph. A similar animation is performed when re–clustering is performed. The locations of the new clusters are calculated by the layout algorithm, which takes into account the previous locations of the nodes comprising the cluster, thus maintaining layout stability.

In order to improve scalability, the synchronization scheme presented in Section 5 can be extended to a hierarchy of synchronization units, which is constructed according to the hierarchical representation of the graph. Each level in the hierarchy contains a synchronization unit. Events are forwarded to the next (higher) level only if they are not contained in the current level in the hierarchy. Using this method, the amount of events reaching the higher levels of the hierarchy (which represent more cores) is significantly reduced. In order to further reduce the volume of events, instead of showing movements of objects using animation, this information can be time-averaged and visualized by changing the frequency of the dashed lines connecting cores.

6.2 Clustering

A clustering algorithm is used in order to compute the hierarchical representation of the graph. The clustering is influenced by the focal nodes, which are interesting nodes selected by the user. A fisheye type effect is used, where nodes farther away from the focal nodes are displayed with less detail. The algorithm, which is summarized in Figure 4, is based on an extension of the agglomerative clustering algorithm.

The algorithm has several inputs. The first is a set of focal nodes (e.g., cores of interest), selected interactively by the user. The second is the distances between nodes, designated $D(u, v)$, which correspond to the weights of edges in the graph. They are calculated according to the frequency of method calls and object moves between cores, as described in Section 4. The third input is the desired number of clusters. The output of the algorithm is a hierarchical clustering of the graph.

In the first step of the algorithm, the shortest distance between each node u and the closest focal node, $D^{focal}(u)$, is calculated. This is done using Dijkstra's algorithm on the focal nodes.

Input: Set of focal nodes; distances between nodes; number of desired clusters

Algorithm:

1. Calculate shortest distance between each node and the closest focus node.
2. Update distances between nodes according to distance to focus node.
3. Perform hierarchical clustering.

Output: Clustering hierarchy of the nodes

Figure 4: Focus-based clustering algorithm

Additionally, the maximum of the minimal distances is computed as $d_{max} = max_{v \in V}\{D^{focal}(v)\}$, where V is the set of nodes in the graph.

In the second step, the distances, $D(u, v)$, between every pair of nodes u, v, are updated according to their proximity to focal nodes. As opposed to the regular fisheye technique, in which geometric distortion is used, our method moves the distortion to the clustering phase. This results in a better layout since the graph is not distorted after layout. We set the initial, joint average distance of nodes u and v and a focus node to

$$D^{focal}_{avg}(u, v) = \frac{D^{focal}(u) + D^{focal}(v)}{2}.$$

It should be noted that the focal node used in $D^{focal}(u)$ may be different from the one used in $D^{focal}(v)$. The distance $D(u, v)$ is distorted to form $D^{distorted}(u, v)$, the updated distance between nodes u and v, according to the following formula:

$$D^{distorted}(u, v) = \frac{D(u, v)}{1 + C \cdot \frac{D^{focal}_{avg}(u,v)}{d_{max}}}.$$

The greater the average distance between the nodes and the closest focal node, the bigger the distortion. This behavior mimics the fisheye effect. Nodes in the periphery are less interesting and therefore have a higher probability of being clustered together, since they are perceived to be close. In our implementation we use $C = 3$. Another option is to have C depend on the size of the graph.

In the last step of the algorithm the actual clustering is performed, using the distances computed in the previous steps. A bottom–up, hierarchical clustering algorithm is used. The algorithm starts

with assigning every node to its own singleton cluster. The algorithm then repetitively greedily joins the two closest clusters. The algorithm terminates when the required number of clusters have been created.

The distance between clusters C_i and C_j is calculated using a modified average distance metric. Only edges (distances) in the set $E_{ij} = \{e = (u, v) \in G | u \in C_i, v \in C_j\}$, that is edges directly connecting a node $u \in C_i$ and a node $v \in C_j$ (e.g., edges crossing the boundary between the clusters) are taken into account. The distance is

$$Dist(C_i, C_j) = \frac{\sum_{(u,v) \in E_{ij}} D^{distorted}(u, v)}{\mid E_{ij} \mid}$$

e.g., the sum of the lengths of the edges divided by the number of edges. This formula is a tradeoff between an exact calculation and a rapid, approximate calculation.

7 Graph Drawing Algorithm

Given a sequence of clustered graphs $G_1, G_2, \ldots, G_n$, our goal is to compute a sequence of graph layouts $L_1, L_2, \ldots, L_n$, so as to adhere as much as possible to the following aesthetic criteria [5, 36, 39, 46]:

- The movement of clusters between successive drawings should be small. Specifically, clusters that are not modified should remain in their previous position if possible. The location of clusters plays an important role in the user's mental map of the graph.
- The change in cluster size between successive drawings should be minimal when the number of vertices in the cluster is similar. Unnecessarily large deviations in size cause the user to be distracted.
- Movement of vertices inside a cluster should be minimized. This improves layout stability.
- The size of each cluster C_i should be proportional to the number of vertices it contains. This allows the user to quickly understand how the mobile objects are distributed between cores.

- In order to conserve screen space, the drawing of each cluster C_i should be compact.
- In order to reduce graph cluttering, overlapping between vertices should be avoided and overlapping between cluster boundaries should be minimal.

Among the different classes of graph drawing algorithms, the force directed algorithm class seems to be the natural choice in our case [8,25,26,46]. Roughly speaking, this approach simulates a system of forces defined on the input graph and outputs a local minimum energy configuration. An edge is simulated by a spring connecting its endpoint vertices. Edge length influences the optimal spring length and edge weight determines its stiffness. The algorithm converges towards a minimum energy position, starting from an initial placement of the vertices.

Our algorithm utilizes the following key ideas. First, dummy vertices and edges are used in order to create a clustered structure. Since clusters are treated as vertices, their motion can be controlled. Second, invisible place-holder vertices are used in order to minimize the movement of clusters and of vertices within clusters. This is done while maintaining compactness and keeping the size of the clusters proportional to the number of vertices they contain. Third, edge length and weight are used as a means of controlling the changes made to the layout. Fourth, to achieve both dynamic stability and distinguish between stable and movable vertices, the set of vertices is partitioned into two sub-sets – stable and movable. The subsets are laid out in a structure that approximates two concentric circles around the center of the cluster. Stable objects are placed in the inner circle and movable objects in the outer one. These ideas are elaborated in this section.

To compute layout L_i, only the last layout, L_{i-1}, and the new graph that needs to be laid out, G_i, are used. This is a fast and simple approach that fits well with the view that incremental layout performs some local changes in the graph. The previous layout is considered a good starting point for the new layout, with some adjustments made according to the changes that occurred.

The first step in computing the new layout, described in Section 7.3, is a merge stage, which merges layout L_{i-1} and graph G_i. In the second stage, an actual layout, L_i^1, is computed using a static force directed layout algorithm with the modifications described in Sections 7.1–7.2. In the third stage, the quality of this layout is checked, as described in Section 7.4. If the layout is deemed satisfactory, it is accepted and $L_i = L_i^1$. Otherwise, a second layout attempt is performed, producing

layout L_i^2. During this attempt, more freedom is given to the layout algorithm in terms of moving vertices, at the expense of weakening the connection between the old and the new layouts. The better of L_i^1 and L_i^2 is selected as the final drawing L_i. The final stage of the algorithm animates the change between the drawings L_{i-1} and L_i in a smooth manner. Figure 5 summarizes the algorithm.

```
procedure incremental_drawing ( L_{i-1}, G_i ) {
    G_i^m = merge_graphs ( L_{i-1}, G_i )
    L_i^1 = layout_graph ( G_i^m )
    if ( density metric of L_i^1 < threshold )
        L_i = L_i^1
    else
        L_i^2 = layout_graph ( modify_graph ( L_i^1 ) )
        L_i = better ( L_i^2, L_i^1 )
    animate_change ( L_{i-1}, L_i )
}
```

Figure 5: Layout algorithm overview in pseudo-code

7.1 Supporting Clusters

In order to visualize the clustered structure of the graph, an invisible dummy attractor vertex is added to each cluster. All of the vertices in the cluster are connected with invisible edges to the attractor vertex [26].

As opposed to existing algorithms, separation between the clusters and meeting the other requirements described above , is accomplished through proper settings of edge lengths and weights. Higher edge weights instruct the underlying force-directed algorithm to try harder to generate edges with lengths close to the optimal lengths supplied to the algorithm.

Five kinds of edge lengths are utilized and indicate the expected level of proximity between their adjacent vertices. The shortest length is assigned to the invisible edges connecting static vertices to the dummy vertex of the cluster they belong to. The edges connecting movable vertices and the

dummy vertex are assigned longer lengths. This creates a layout that resembles two concentric circles. The next type of edge is the edge between vertices. If both vertices at the endpoints of the edge are contained in the same cluster, a shorter length is set than if the vertices are in different clusters. This increases the separation between clusters. The last kind of edges are cluster-cluster edges. The length of these edges is variable and depends on the requested proximity between the different clusters, which is determined by the application, e.g., by the amount of interaction between clusters.

Inter-cluster edges are assigned lower weights than intra-cluster edges. This is done in an attempt to give inter-cluster edges less influence on the layout. This is important when vertices move between clusters. In such cases, it is preferable to stretch or shorten the length of the edges somewhat, rather than displace vertices.

In our implementation, the lengths assigned to the edges connecting a static vertex to a dummy vertex, a movable vertex to a dummy vertex, two regular vertices in the same cluster and two regular vertices located in different clusters, are 1, 2, 1.5 and 4 units of length, respectively. The lengths assigned to cluster-cluster edges vary between 5 and 6 units, where the dummy vertices are used as endpoints for cluster-cluster edges. The weight of intra-cluster edges is set to 1 unit and the weight of inter-cluster edges is set to 2.5 units. These values represent a compromise between stable layouts to aesthetic ones. Allowing the user control over these parameters will tailor the visualization to the user's preferences.

7.2 Minimizing Visual Changes

Invisible vertices, called *spacer* vertices, are added to each cluster, in an attempt to reduce the change in clusters' outlines and minimize the movement of clusters between successive layouts.

The spacer vertices are used as place-holders for regular vertices in a cluster. Like regular vertices, they are connected with invisible edges to the dummy vertex of the cluster to which they belong. When a vertex is removed from a cluster, a spacer vertex is added to the cluster instead of it. The initial location of the spacer vertex is set to be the location of the vertex that left the cluster. This is done in order to keep the size of the cluster constant and in order to reserve space for a

new vertex that might be added to the cluster in the future. When a vertex moves (or is added) to a cluster, the spacer vertex that is closest to its previous location is replaced by this new vertex.

However, when adding or removing spacers, the algorithm keeps the number of spacers in a cluster between an upper and a lower fraction of the number of vertices in the cluster. This is done in order to give the algorithm breathing room when modifying clusters. Moreover, the limits are set so as to avoid a case in which a cluster with a very small number of regular, visible vertices occupies a large area due to the many spacer vertices it contains.

When calculating the outline of each cluster, which is often simply the bounding box, the spacer vertices are taken into account as if they were regular visible vertices. Obviously, this minimization of the movements comes at the expense of extra screen space, which is occupied by the spacers.

7.3 Merging Graphs

The first step in performing the incremental layout is merging the new graph to be drawn, G_i, and the previous graph drawing, L_{i-1}. The result of the merge stage is a partially laid out graph, G_i^m, in which some of the vertices are assigned initial coordinates. After merging, the graph G_i^m is laid-out by the static layout algorithm. The quality of the resulting incremental layout depends on the initial conditions computed by the merging algorithm.

Merging is performed in several steps. Unchanged and dummy vertices are assigned initial coordinates from L_{i-1}. Then, clusters to which vertices were both added and removed are handled. The added and removed vertices of a cluster are paired-up, and the initial coordinates of an added vertex is set to the coordinates of a removed vertex.

Then, vertices that were added to a cluster or removed from it, but cannot be paired-up, are handled, as discussed in Section 7.2. Next, the vertices in new clusters (who exist in G_i but not in L_{i-1}), are inserted into the graph without initial coordinates, along with new spacer vertices. The number of the latter is set to a constant fraction of the number of vertices in the cluster.

The last stage of merging involves vertex pinning, which restricts vertex movement, allowing it to move only as an indirect result of the movement of an unpinned vertex. We have experimented with several strategies for computing the set of vertices to be pinned. Our conclusion is that pinning

all vertices that were assigned coordinates achieves good results in terms of layout stability.

7.4 Avoiding bad local minima

One of the disadvantages of force directed layout is that it converges to a local minimum of graph energy. In some cases, this leads to an unsatisfactory layout. In order to detect such conditions, we define, for cluster C_i,

$$density\ metric(C_i) = \frac{area(bounding\ box(C_i))}{number\ of\ vertices(C_i)}.$$

For the entire graph G we define $density\ metric(G) = max_{C_i \in G}\{density\ metric(C_i)\}$.

Experience has shown that a correlation exists between high density metric values and unaesthetic layouts, including ones that contain overlaps between clusters.

When the graph density metric exceeds a threshold, a second layout, L_i^2, is computed. The restrictions on vertex movement are relaxed – vertices are not pinned down. This gives the layout algorithm more freedom and allows it to converge to a better result. The layout algorithm is re-run with the positions of the vertices in L_i^1 as the initial condition. The new layout L_i^2 still resembles L_i^1 because of the supplied initial condition. The final layout is selected as the layout with the lower density metric between L_i^1 and L_i^2. This demonstrates the tradeoff between preserving the mental map and creating an aesthetically pleasing layout.

8 Implementation

MOVIS was implemented on top of *FarGo* [22], a Java-based mobile object framework. FarGo contains extensive monitoring facilities [40] and uses a source–to–source compiler called *Fargoc* for generating proxies and other code used to implement support for mobile objects. Our implementation is Java based. We use the Java3D API for generating the visualization.

Our system is composed of several components. In each core (machine), a special *local profiling object*, used to collect events, is instantiated. This object listens both to events generated by the Fargo monitor and to events generated by our modified Fargoc compiler. The events generated

by each core are forwarded to a main *event collection object*. This object either stores the events for offline visualization or forwards them to the event synchronization unit, described Section 8.2. After creating a synchronized event list, from which a consistent run of the application can be constructed, the events are sent to the visualization component. Events generated by the user, such as requests for re-clustering or zoom in / zoom out operations are fused together with the events collected from the system, in order to form a unified event queue that is visualized.

8.1 Event Generation

One of the goals of a program visualization system is to generate events with minimal effort by the programmer and the user of the application being visualized, while perturbing the running application as little as possible. In this section we describe how this is achieved.

The interesting events are related to communication between mobile objects and movement of objects between cores. Since location transparency needs to be maintained when communication is performed between mobile objects, some kind of proxy needs to be used in order to forward the method call to the actual destination object. This proxy is generated either statically [22] or dynamically [1,42]. This is where the event generation code is (automatically) inserted.

In order to trace method calls, the Fargoc compiler was modified to transparently generate an event each time execution enters an interface method of a mobile object. Generating events for movement of objects between cores is implemented by piggybacking onto the migration code supplied by the middleware. Other types of actions for which events need to be generated include the creation and destruction of mobile objects and cores (e.g., connecting/disconnecting from the application network). This is handled by tapping into an existing profiling interface.

8.2 Event Synchronization Component

The event synchronization component receives events from all of the event collection objects located at the different cores that constitute the application to be monitored. It reorders the events in order to generate a sequence of events that is consistent. This stream of events is then visualized.

The implementation of the synchronization component follows several rules and observations made in this paper. The first is that all events generated at a core are reported in FIFO order and each event depends on the previous event. The second is that a send event should be reported (to the visualization) before (depends on) the receive event. The algorithm is described in Figure 6.

```
procedure handle_event (event e) {
    if (e's core is blocked)
        queue_event(e)
    else
        if (e is a send or internal event)
            commit_event(e)
        else //e is a receive event
            if (e depends on a committed event)
                commit_event(e)
            else
                queue_event(e)
}
procedure commit_event(event e) {
    send_to_vis(e)
    if (e can unblock a core)
        BFS_unblock_core(e.getCore())
}
```

```
procedure BFS_unblock_core(core c) {
    active_cores_list ← {c}
    while (active_cores_list ≠ ∅)
        c = remove_first(active_cores_list)
        if (c has more queued events)
            e_c = c.nextEvent()
            if (e_c can be sent)
                send_to_vis(e_c)
                if (e_c is a send event)
                    dest(e_c) = e_c destination core
                    if (dest(e_c) blocked on e_c)
                        add dest(e_c) to active_cores_list
                add c to active_cores_list
}
```

Figure 6: Event synchronization algorithm

For each core, the synchronization component maintains a queue of events. This queue contains received events that cannot be forwarded to the visualization component, since a dependent send event was not received yet by the synchronization component. We will call the act of sending an event to the visualization component *committing the event*. Committing a send event may be delayed since it in itself is dependant on a previous event that has not been committed, yet.

When a new event is received by the synchronization component, the following is done. First, a check is made if the core from which the event was sent is blocked, e.g. waiting for events. If this is the case, the event is added at the end of the event queue of the core. If the core is not queuing events and the event is a send event - it is committed. A check is made if there is another core that is blocked on this event. If this is the case, events from the blocked core may be committed, according to their order in the queue. If the newly received event is a receive event, a check is made to determine if the send event that it depends on was already sent. If this is the case, the event is committed. If this is not the case, the event is queued and its core enters the blocked state.

When a core unblocks, the queued events are committed. This, in turn may cause other cores to become unblocked (due to committing a send event that the blocked core depends on). A list of active cores is maintained. Each time one event is committed from a core, the activity switches over to the next core in the list. This is similar to advancing in a graph using a BFS algorithm. The motivation of using this method is to create a stream of events that will produce animation that is maximally parallel. Switching from one core to an other while committing events attempts to expose the possible parallelism to the visualization component. The synchronization component can be modified to produce a variety of interesting orderings, as described in [29].

9 Results

Our system has been used for visualizing several applications, including a mobile object simulator, an e-commerce application [24] and a distributed e-mail system (abbreviated DEM) [2]. We first present visualizations of our mobile object simulator and then proceed to discuss the application of MOVIS to the DEM system.

Mobile object simulator In order to test our visualization system, a mobile object simulator was implemented. The simulator uses a configuration file which governs the activities of mobile objects it creates. The number of objects, their creation and destruction time and location, their movement and communication patterns are all specified in the configuration file. In the next paragraphs we use a sample run of the simulator in order to demonstrate the quality of our layout algorithm.

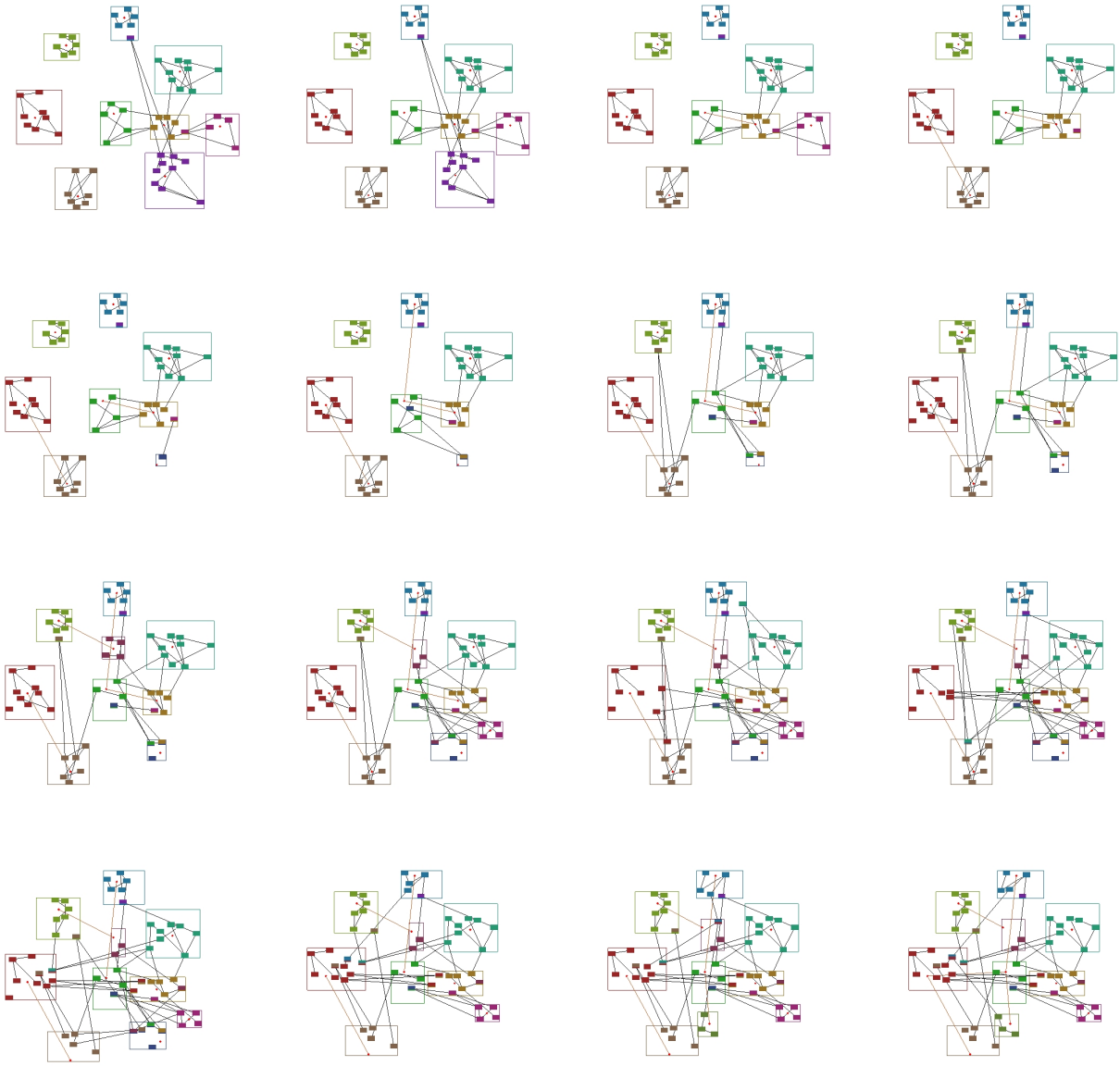
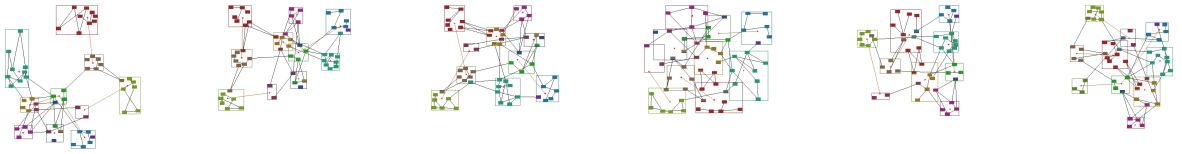


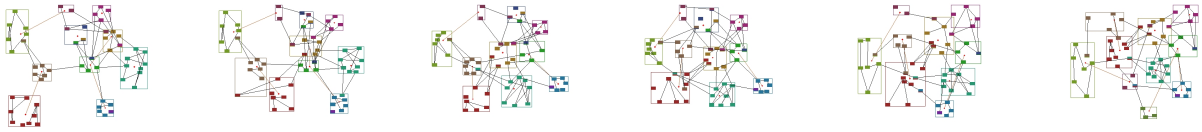
Figure 7: Sample animation sequence (from left to right and top to bottom)

Figure 7 demonstrates an animation sequence created with our visualization algorithm. To measure the quality of the resulting layouts, we identify several criteria. The *density metric*, discussed in Section 7.4, is used to measure the compactness of the layout. The *sum of displacement of clusters between each pair of successive layouts*, is used to measure the stability of the layout. The *percentage of clusters with the same size between successive layouts* helps demonstrate the effectiveness of using spacer vertices in minimizing visual changes to the graph.

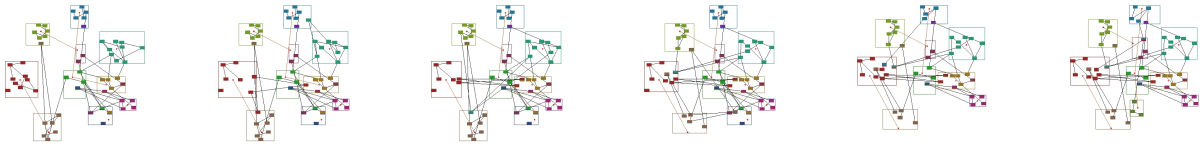
The performance of our algorithm is compared to two other algorithms. The first is a non-incremental algorithm which computes each layout from scratch using force-directed methods. The second is a variant of our incremental algorithm in which vertices are assigned initial coordinates computed in the merge stage, but vertex pinning and spacer vertices are not used. We use this second algorithm in order to show that simply reusing the initial coordinates from the previous graph does not yield satisfactory results. Figure 8 shows a comparison of the layouts computed by the three algorithms. Note that only our algorithm manages to compute stable layouts.



(a) Non-incremental layout



(b) Incremental layout without using pinning and spacers



(c) Our incremental layout

Figure 8: Comparing the three layout algorithms

Table 1 summarizes the average values of each of the above metrics on the sequence in Figure 7. All algorithms produce similar cluster densities. The cluster displacement of our algorithm is by far superior to the non–incremental algorithm, averaging about one thirtieth of the non-incremental algorithm. Reducing the movement of clusters has indeed been one of the main design goals of the algorithm. While the non-incremental algorithm frequently modifies the size of clusters between layouts, the incremental algorithm manages to conserve the size of clusters almost eighty percent of the time. This is facilitated by the spacer vertices that are used to minimize visual changes to

the graph. As seen in the table, the other incremental algorithm, which does not use vertex pinning and spacer nodes produces results which are better than the non–incremental algorithm and worse than our incremental algorithm. Finally, the running time of the incremental algorithm is about 1.5 times the running time of the non-incremental algorithm, which is reasonable.

In summary, it has been demonstrated that the algorithm computes a compact and space efficient graph layout, while minimizing the displacement and changes to clusters between layout iterations. This is turn helps the user maintain the mental map of the mobile object system.

Average / Algorithm	non-incremental	no vertex pinning	with vertex pinning
density metric [$area/vertices$]	$1.8557 * 10^4$	$1.7269 * 10^4$	$1.5316 * 10^4$
cluster displacement [distance]	13.7	5.53	0.339
fraction of clusters with the same size	0.048	0.104	0.770
running time [ms.]	260	573	368

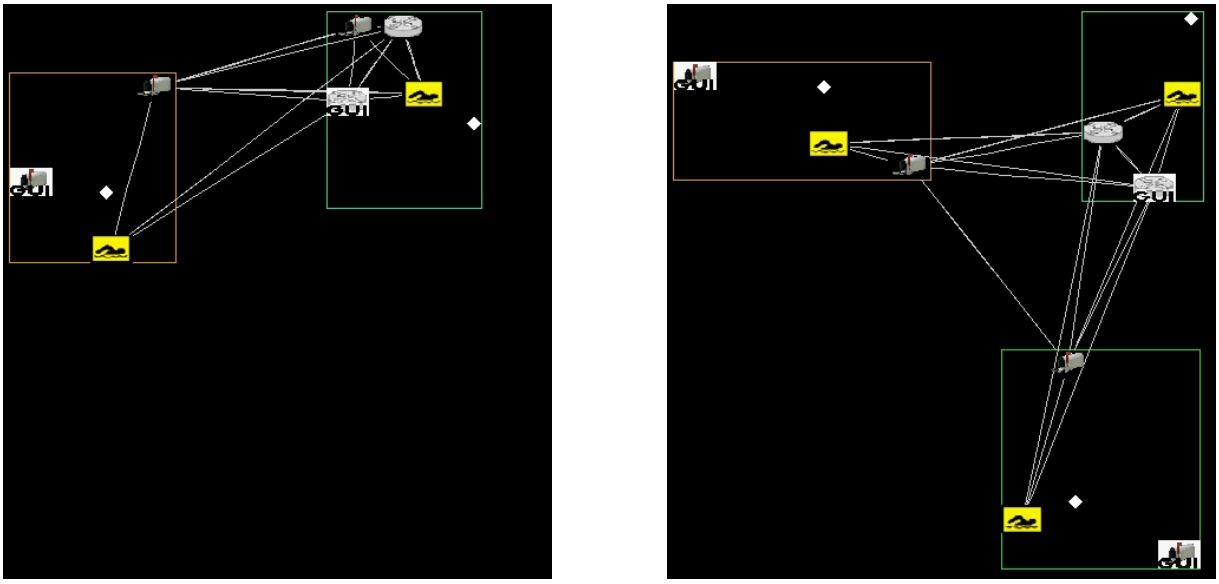
Table 1: Average results of an animation sequence

Mobile-object based E-mail application E-mail is one of the most popular Internet applications. Nowadays, e-mail architectures are governed by a server-centric design, which implies a handful of weaknesses such as a single point of failure, storage and processing stress, bottlenecks and inefficiency.

The goal of the DEM system is to overcome these drawbacks. The service is provided through the use of the participants' resources. Lightweight servers and users' mailboxes all scatter between participants' computers instead of residing on a single server (or cluster). Through the use of the mobile objects paradigm, the mailboxes and servers are able to travel on the "live" network, so that they continue their operation despite the fact that participants constantly join and leave the network. Most of the communication is done directly between users, thus removing the bottlenecks caused by mail servers. The system's components are replicated across numerous hosts, eliminating single point of failure problems. Storage and processing stress is reduced as participants take an even share of the burden. All of this yields a reliable and scalable system, with negligible operational and maintenance cost.

Visualization has been used during the development of this application – for debugging purposes as well as for managing and monitoring its deployment across the network. Due to the complexity of the architecture, its developer expressed a need for visualization at the very early stages of implementation. Using visualization, several problems were quickly discovered. For example, a case where an object does not flee from a core that is shutting down was uncovered.

In this application, icons have been used to represent the objects, instead of the default rectangles. The mailboxes are displayed using a mailbox icon. Servers are represented as gray disks. Yellow pools represent mailbox placeholders. Finally, the GUI is represented by a mailbox icon with a white background.



(a) Before movement (b) A new core was created and a mailbox migrated to it

Figure 9: Mailbox mobility in the DEM system

Figure 9 shows a visualization of the movement of a mailbox between computers. In Figure 9(a) there is one mailbox in each core. In Figure 9(b) a mailbox moved to a new core that connected to the service, shown at the bottom.

Filtering of method calls was used in order to show specific interesting events. For example, Figure 10 shows an e-mail message being sent from the source mailbox directly to the destination mailbox. The message, in transit, is drawn inside a red circle. An accompanying movie can be found at http://www.ee.technion.ac.il/∼ayellet/Movies/FrishmanTal.mov.

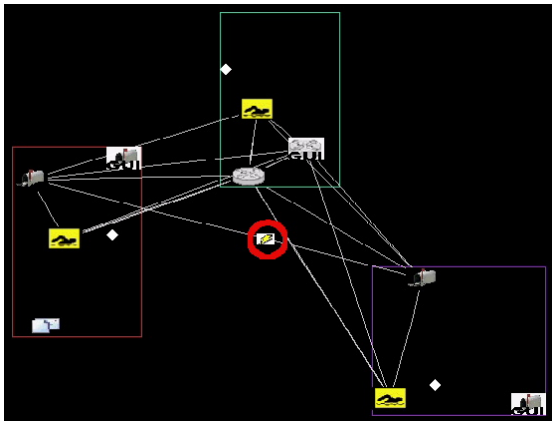


Figure 10: Sending an e-mail in the DEM system

10 Conclusion

We have presented MOVIS – a system for visualizing mobile object frameworks. The key features of these frameworks – object mobility, location transparency, and distributed operation – are addressed by our system. A clustered graph is used to concurrently show the physical connections between cores and the logical connections between objects. A clustering algorithm, which is influenced by the areas of interest to the user, is used to provide a hierarchical, scalable context+focus visualization. The overall complexity of the graph is user controlled. The visualization is dynamic: incremental graph layout and animation are used to depict changes in a smooth, comprehensible manner.

MOVIS has been used for monitoring and debugging as well as for presenting system architectures. It has been used in several scenarios, including simulators, e-commerce and distributed e-mail.

There are several avenues of future research. Additional levels of detail can be integrated into the visualization. The existing profiling infrastructure can be used to supply object-specific information such as memory usage and creation time. Information about the cores themselves, such as thread count, memory usage and CPU usage can also be integrated into the visualization.

Acknowledgment

This work was partially supported by the European FP6 NoE grant 506766 (AIM@SHAPE),by the Technion V.P.R Fund, and by the Ollendorff Foundation.

References

[1] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A language for resource-aware mobile programs. In *Mobile Object Systems: Towards the Programmable Internet*, number 1222 in LNCS, pages 111–130, 1996.

[2] S. Bercovici, Y. Frishman, I. Keidar, and A. Tal. Decentralized electronic mail. In *International Workshop on Dynamic Distributed Systems (IWDDS)*, 2006.

[3] F. Bertault and M. Miller. An algorithm for drawing compound graphs. In *Proc. 7th Int. Symp. Graph Drawing (GD 1999)*, number 1731 in LNCS, pages 197–204, 2000.

[4] U. Brandes, D. Fleischer, and T. Puppe. Dynamic spectral layout of small worlds. In *Proc. 13th Int. Symp. Graph Drawing, GD*, pages 25–36, 2005.

[5] S. S. Bridgeman and R. Tamassia. A user study in similarity measures for graph drawing. *J. Graph Algorithms Appl*, 6(3):225–254, 2002.

[6] S. K. Card, J. D. Mackinlay, and B. Shneiderman, editors. *Readings in Information Visualization Using Vision to Think*. Morgan Kaufman, 1999.

[7] J. H. Chuang, C. C. Lin, and H. C. Yen. Drawing graphs with nonuniform nodes using potential fields. In *Proc. 11th Int. Symp. Graph Drawing (GD 2003)*, number 2912 in LNCS, pages 460–465, 2004.

[8] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *Proc. ACM Symposium on Software Visualization*, pages 77–86, 2003.

[9] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: An annotated bibliography. *Computational Geometry: Theory and Applications*, 4(5):235–282, 1994.

[10] S. Diehl and C. Gorg. Graphs, They Are Changing - Dynamic Graph Drawing for a Sequence of Graphs. In *Proc. 10th Int. Symp. Graph Drawing (GD 2002)*, number 2528 in LNCS, pages 23–31, 2002.

[11] P. Eades and Q. W. Feng. Multilevel visualization of clustered graphs. In *Proc. 4th Int. Symp. Graph Drawing (GD 1996)*, number 1190 in LNCS, pages 101–112, 1996.

[12] N. Elmqvist and P. Tsigas. Growing squares: animated visualization of causal relations. In *Proc. ACM Symposium on Software Visualization*, pages 17–26, 2003.

[13] C. Erten, P. J. Harding, S. G. Kobourov, K. Wampler, and G. V. Yee. GraphAEL: Graph animations with evolving layouts. In *Proc. 11th Int. Symp. Graph Drawing*, pages 98–110, 2003.

[14] Y. Frishman and A. Tal. Dynamic drawing of clustered graphs. In *Proceedings of the IEEE Symposium on Information Visualization, InfoVis*, pages 191–198, 2004.

[15] Y. Frishman and A. Tal. Visualization of mobile object environments. In *ACM Symposium on Software Visualization*, pages 145–154, 2005.

[16] G. W. Furnas. Generalized fisheye views. In *Human Factors in Computing Systems, CHI'86 Conference Proceedings*, pages 16–23, 1986.

[17] E. R. Gansner and S. C. North. Improved force-directed layouts. In *Proc. 6th Int. Symp. Graph Drawing (GD 1998)*, number 1547 in LNCS, pages 364–373, 1998.

[18] Carsten Görg, Peter Birke, Mathias Pohl, and Stephan Diehl. Dynamic graph drawing of sequences of orthogonal and hierarchical graphs. In *Proc. 12th Int. Symp. Graph Drawing, GD*, volume 3383 of *LNCS*, pages 228–238, 2004.

[19] S. Hachul and M. Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In *Graph Drawing*, pages 285–295, 2004.

[20] D. Harel and Y. Koren. A Fast Multi-Scale Algorithm for Drawing Large Graphs. *J. Graph Algorithms Appl.*, 6(3):179–202, 2002.

[21] D. Harel and Y. Koren. Drawing graphs with non-uniform vertices. In *Proc. Working Conference on Advanced Visual Interfaces (AVI'02)*, pages 157–166. ACM Press, 2002.

[22] O. Holder, I. Ben-Shaul, and H. Gazit. Dynamic layout of distributed applications in fargo. In *Proc. International Conference on Software Engineering*, pages 163–173, 1999.

[23] M. L. Huang and P. Eades. A fully animated interactive system for clustering and navigating huge graphs. In *Proc. 6th Int. Symp. Graph Drawing (GD 1998)*, number 1547 in LNCS, pages 374–383, 1998.

[24] A. Joseph, R. Dar, and Y. Almog. *Active Market Project Report*, 2000. Available at `http://tochna.technion.ac.il/project/amarket/html/home.htm`.

[25] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, April 1989.

[26] M. Kaufmann and D. Wagner, editors. *Drawing Graphs: Methods and Models*. Number 2025 in Lecture Notes in Computer Science, LNCS. Springer-Verlag, 2001.

[27] J. A. Kohl and G. A. Geist. The PVM 3.4 tracing facility and XPVM 1.1. In *Proc. Hawaii International Conference on System Sciences*, volume 1, pages 290–299, 1996.

[28] Y. Koren, L. Carmel, and D. Harel. Drawing huge graphs by algebraic multigrid optimization. *Multiscale Modeling & Simulation*, 1(4):645–673, 2003.

[29] E. Kraemer and J. T. Stasko. Creating an accurate portrayal of concurrent executions. *IEEE Concurrency*, 6(1):36–46, 1998.

[30] E. Kraemer and J.T. Stasko. The visualization of parallel systems: an overview. *Journal of Parallel and Distributed Computing*, 18(2):105–117, 1993.

[31] G. Kumar and M. Garland. Visual exploration of complex time-varying graphs. *IEEE Trans. on Visualization and Computer Graphics, Proc. InfoVis*, 2006.

[32] L. Lamport. Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*, pages 558–565, July 1978.

[33] Yi-Yi Lee, Chun-Cheng Lin, and Hsu-Chun Yen. *Mental Map Preserving Graph Drawing Using Simulated Annealing*, volume 60 of *Conferences in Research and Practice in Information Technology*. 2006.

[34] K. A. Lyons, H. Meijer, and D. Rappaport. Algorithms for cluster busting in anchored graph drawing. *J. Graph Algorithms and Applications*, 2(1):1–24, 1998.

[35] D. Milojicic, F. Douglis, and R. Wheeler, editors. *Mobility: Processes, Computers and Agents*. ACM Press, 1999.

[36] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, 6(2):183–210, 1995.

[37] J. Moe and D. A. Carr. Understanding distributed systems via execution trace data. In *International Workshop on Program Comprehension*, pages 60–69, 2001.

[38] Y. Moses, Z. Polunsky, A. Tal, and L. Ulitsky. Algorithm visualization for distributed environments. *Journal of Visual Languages and Computing*, 15(1):97–123, 2004.

[39] S. C. North. Incremental layout in dynadag. In *Proc. 3rd Int. Symp. Graph Drawing (GD 1995)*, number 1027 in LNCS, pages 409–418, 1995.

[40] O. Holder and I. Ben-Shaul and H. Gazit . System support for dynamic layout of distributed applications. In *19th International Conference on Distributed Computing Systems (ICDCS'99)*, pages 403–411, 1999.

[41] Object Management Group. *The Common Object Request Broker: Architecture and Specification. Revision 2.2*, February 1998.

[42] ObjectSpace. *ObjectSpace Voyager Core Package: Technical Overview*, December 1997.

[43] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the execution of java programs. In *Proceedings of the International Seminar on Software Visualization*, number 2269 in LNCS, pages 151–162, 2001.

[44] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. Scalable Parallel Libraries Conference*, pages 104–113, 1993.

[45] J. T. Stasko and E. Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, 1993.

[46] I. G. Tollis, G. Di Battista, P. Eades, and R. Tamassia. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[47] Tom Sawyer graph layout toolkit, 2004. Currently Available at http://www.tomsawyer.com.

[48] B. Topol, J. T. Stasko, and V. Sunderam. Pvanim: A tool for visualization in network computing environments. *Concurrency: Practice and Experience*, 10(14):1197–1222, 1998.

[49] C. Walshaw. A Multilevel Algorithm for Force-Directed Graph Drawing. *J. Graph Algorithms Appl.*, 7(3):253–285, 2003.

[50] X. Wang and I. Miyamoto. Generating customized layouts. In *Proc. 3rd Int. Symp. Graph Drawing (GD 1995)*, number 1027 in LNCS, pages 504–515, 1996.

[51] Y. Wang and T. Kunz. Visualizing mobile agent executions. In *Second International Workshop on Mobile Agents for Telecommunication Applications (MATA 2000)*, number 1931 in LNCS, pages 103–114, 2000.

[52] R. Wiese, M. Eiglsperger, and M. Kaufmann. yfiles: Visualization and automatic layout of graphs. In *Proc. 9th Int. Symp. Graph Drawing (GD 2001)*, number 2265 in LNCS, pages 453–454, 2001.

[53] A. Wong, T. Dillon, M. Ip, and W. Lin. A generic visualization framework to help debug mobile-object-based distributed programs running on large networks. In *WORDS*, pages 240–250. IEEE Computer Society, 2001.