# Visualization of Graph Representations of Data-Flow Programs

VICTOR KASYANOV, ELENA KASYANOVA, TIMUR ZOLOTUHIN
Institute of Informatics Systems
Novosibirsk State University
Novosibirsk, 630090
RUSSIA
kvn@iis.nsk.su

*Abstract:* - Information visualization plays an important role in human life. It is believed that about 90% of all received information people receive through vision. Humanity for thousands of years has overcome the way from simple imaging methods in the form of rock paintings to maps, charts and graphs. Currently, visualization based on graph models is an inherent part of the processing of complex information about the structure of objects, systems and processes in many applications in science and technology. In this paper, we describe an effective algorithm for visualization of graph representations of data-flow programs and its effective implementation within the Visual Graph system for visualization of arbitrary attributed hierarchical graphs with ports.

*Key-Words:* - Attributed graph, hierarchical graph, graph with ports, graph drawing, visualization of data-flow program

## 1 Introduction

Visualization is a process of transformation of large and complex abstract forms of information into visual form, strengthening user's cognitive abilities and allowing them to take the most optimal decisions. Graphs are the most common abstract structure encountered in computer science and are widely used for abstract information representation.

Currently, visualization based on graph models is an inherent part of the processing of complex information about the structure of objects, systems and processes in many applications in science and technology [1 - 4]. At the market there are widely presented software products, using the information visualization on the basis of graph models: such as aiSee [5], yEd [6], Cytoscape [7], Higres [8] or Visual Graph [9].

The Cloud Parallel Programming System (CPPS) being under development at Institute of Informatics system is a visual programming system on the base of the Cloud Sisal language [10]. The Cloud Sisal language continues the tradition of previous versions of the SISAL language, remaining a functional data-flow programming language oriented to writing large scientific programs, and expands their capabilities with tools for supporting cloud computing [11].

The goal of the CPPS project is to develop methods and tools for architecturally independent parallel programming to support cloud-based high-performance computing (supercomputing) based on the functional-data-flow paradigm and the transformational approach. It is assumed that the CPPS system will provide means to write and debug Cloud-Sisal-programs regardless target architectures on low-cost devices. Adaptation of an architecturally independent functional Cloud-Sisal-program to a specific parallel supercomputer will be implemented by optimizing cross-compilation during which the program is subjected to the necessary optimizing and restructuring transformations under user control. By this the degree of the user's participation can be different — from complete non-participation or simply providing additional information about the program and the supercomputer (in the form of annotations-statements) to direct or indirect (using annotations-directives) control of the transformations. In particular, the user should be able to visually manipulate the Cloud Sisal program within the framework of its graph representation. As a result of cross-compilation an effective parallel program, appropriate to a target execution platform, will be constructed and can be executed in clouds.The CPPS system uses an internal graph representation of the Cloud Sisal programs, which focuses on their semantic and visual processing and which is based on attributed hierarchical graphs [12]. This representation, called Intermediate Representation (IR), in contrast to the control flow graph, commonly used in optimizing compilers for imperative languages (such as C or Fortran), expresses data dependencies, with control left implicit [10].

The CPPS system will support visualisation of IR-graphs of Cloud-Sisal programs and its use in development and debugging of architecturally independent functional Cloud-Sisal-programs. The system will also support visualization of both internal data structures that arise in the optimizing cross-compiler when building parallel programs, and the dynamic processes that arise when executing parallel programs. This visualization can be used for control of optimizing compilation in order to improve the performance of parallel programs obtained from their functional specifications by using the cross-compiler.

The main difficulties in solving problem of the visualization of IR graphs are due to the fact that, in contrast to the standard problem of drawing of graphs on the plane [1 - 4] vertices of IR-graphs are connected by arcs through their different ports as well as can have different sizes depending on the image characteristics of those graphs, which are embedded in these vertices.

In this paper, we describe an effective algorithm for visualization of IR-graphs and its effective implementation within the framework of Visual Graph system for visualization arbitrary attributed hierarchical graphs with ports.

The rest of the paper is structured as follows. Section 2 presents definitions of hierarchical graphs and graph models. IR-graphs and its drawings are presented in Section 3. Section 4 is a general description of algorithm. Section 5, 6 and 7 presents in more details three main stages of the algorithm: layer assignment when vertices to horizontal layers are assigned and thus their *y*-coordinates are determined, crossing reduction when orders of vertices within each layer to reduce the number of arc crossings are determined, horizontal coordinate assignment when an *x*-coordinates for each vertex is determined. Implementation of the algorithm within the Visual Graph system is presented in Section 8. Section 9 provides our conclusion.

## 2 Hierarchical Graphs and Graph Models

Let us consider some definitions from [12, 13].

Let $G$ be a graph of some type, e.g. $G$ can be an undirected graph, a digraph or a hypergraph. A graph $C$ is called a *fragment* of $G$, denoted by $C \subseteq G$, if $C$ includes only elements (vertices and edges) of $G$. A set of fragments $F$ is called *a hierarchy of nested fragments* of the graph $G$, if $G \in F$ and $C_1 \subseteq C_2$, $C_2 \subseteq C_1$ or $C_1 \cap C_2 = \varnothing$ for any $C_1$, $C_2 \in F$.

A *hierarchical graph* $H = (G, T)$ consists of a graph $G$ and a rooted tree $T$ that represents an immediate inclusion relation between fragments of a hierarchy $F$ of nested fragments of $G$. $G$ is called the *underlying graph* of $H$. $T$ is called the *inclusion tree* of $H$.

A hierarchical graph $H$ is called a *connected* one, if each fragment of $H$ is connected graph, and a *simple* one, if all fragments of $H$ are induced subgraphs of $G$.

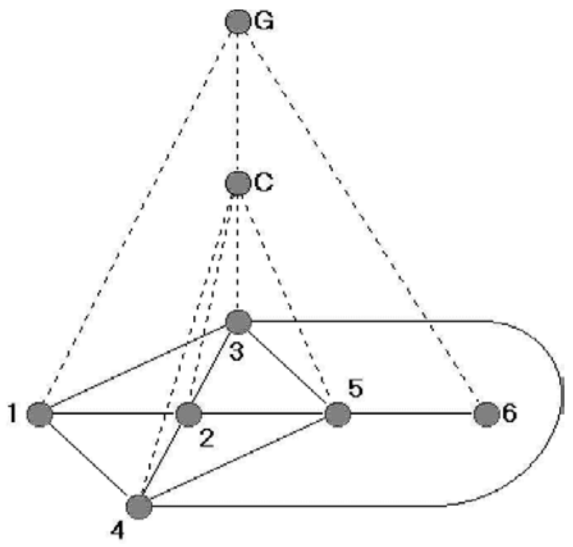

Fig. 1. A simple hierarchical graph $H=(G, T)$ which has only two nontrivial fragments: $G$ and $C$.

It should be noted that any clustered graph $H$ [14] can be considered as a simple hierarchical graph $H = (G, T)$, such that G is an undirected graph and the leaves of $T$ are exactly the trivial subgraphs of $G$ (See Fig. 1).

Let $V$ be a set of objects called simple *labels* (e.g. $V$ can include some numbers, strings, terms and graphs). Let $W$ be a set of *label types* of graph elements and let a label set $V(w) = V_1 \times V_2 \times \ldots \times V_s$, where $s \geq 1$ and for any $i$, $1 \leq i \leq s$, $V_i \subseteq V$, be associated with each $w \in W$.

*A labeled hierarchical graph* is a triple $(H,M,L)$, where $H$ is a hierarchical graph, $M$ is *a type function* which assigns to each element (vertex, edge and fragment) $h$ of $H$ its type $M(h) \in W$, and $L$ is a *label function*, which assigns to each element $h$ of $H$ its label $L(h) \in V(M(h))$.

Under the graph model, in general, we understand a class of graph objects being attributed (labeled) graphs with a given equivalence relation on it. So in the definition of a graph model we can distinguish between a static (or syntactic) part of the specification which defines a class of graph objects, and a dynamic (or semantic) part which defines a partition of this class into subclasses of graphs being pair-wise equivalent.

When the image of a graph model is made a type of its elements may be associated with a certain geometrical shape of corresponding representations and / or with their certain color range, as well as with the place and manner of representations of attributes relating to the elements of the type.

As for the dynamic part of a hierarchical graph model, it brings to the visualization of graph models different aspects of animation. Two main approaches to the presentation of semantic part of the graph model are used typically: either by explicitly specifying a set of so-called invariants (i.e. properties being common to all equivalent models), which distinguishes the equivalence classes of graph models, or through so-called equivalent transformations of graph models, which retain the specified set of invariants. Both approaches to the presentation of the semantic graph model are based on graph transformations and actively studied and developed in the theory of program schemes (see, for example, [15]).

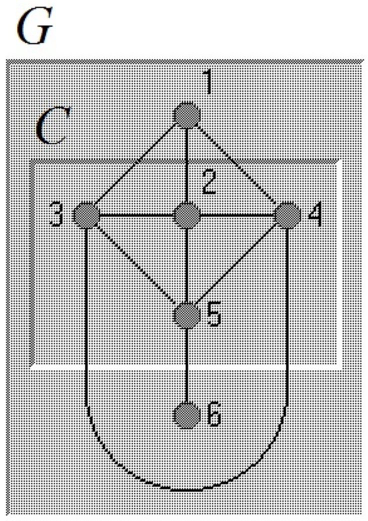

Fig. 2. A drawing of the simple hierarchical graph $H$=($G$, $T$) with two nontrivial fragments $G$ and $C$ represented as rectangles.

A *drawing D* of a hierarchical graph $H$ = ($G$,$T$) is a representation of $H$ in the plane such that the following properties hold (See Fig. 2).

- Each vertex of $G$ is represented either by a point or by a simple closed region. The region is defined by its *boundary* - a simple closed curve in the plane.
- Each fragment of $G$ is drawn as a simple closed region which includes all vertices and subfragments of the fragment.
- Each edge of $G$ is represented by a simple curve between the drawings of its endpoints.

So, different fragments in a drawing of a hierarchical graph can have different sizes depending on the image characteristics of those graphs, which are included in these fragments.

## 2 IR-Graphs and Their Drawings

The CPPS system uses so-called IR-graph as an internal graph representation of a source Cloud-Sisal-program. The vertices of the IR-graph correspond to the expressions of the Cloud Sisal program, and the arcs show the transmissions of data between the vertex ports, the ordered sets of which are assigned to the vertices as their arguments (input ports or inputs) and results (output ports or outputs).

Vertices of the IR-graphs denote operations on their inputs (arguments), the results of which are at the outputs of vertices. There is, however, a special kind of vertices denoting literals (constants) of any type, each of which has one output and an empty set of inputs.

Vertices of the IR-graphs can be either simple or compound. Simple vertices do not have an internal structure in addition to the associated operation, such as add or divide. The compound vertices (or fragments) correspond to complex expressions of Cloud-Sisal-program, such as loop expression or function, and contain ordered sets of vertices (or subfragments) corresponding to the subexpressions from which they consist. The number of subfragments may be fixed (in Loop and Let vertices), may vary (in Function and Select vertices).

Because of the property of the Cloud Sisal language, any IR-graph is a DAG (Directed Acyclic Graph) and does not contain two arcs that enter the same input port.

The following rules for drawing of IR-graphs are assumed:

1. Simple vertices without inputs are represented in the form of circles containing the representations of the constants.

2. Simple vertices with inputs are represented in the form of rectangles with semicircular projections from above, representing the vertex inputs in their order from left to right, and semicircular projections from below, depicting the outputs of the vertex.

Inside the rectangles are the representations of corresponding operations.

3. Compound vertices are represented as rectangles with a rectangular ledge from the top left to indicate the type of the composite vertex, as well as with circles from the top right and bottom to show the inputs and outputs of this vertex in their order from left to right. Each circle representing a certain pole of the fragment consists of a semicircular projection outward and a semicircular projection inside this rectangle. Inside the rectangle that represents the compound vertex, there are images of all the vertices and all arcs contained in it, and only they (See Fig. 3).

4. Images of two different vertices either do not intersect, or one of them lies entirely in the other.

5. Arcs are represented as curves (splines) with arrows that connect the corresponding ports and do not intersect the vertices' images at their internal points

So, IR-graphs are hierarchical graphs whose vertices are connected by arcs through their different ports, and the problem of visualization of IR graphs cannot be solved by methods and algorithms known for drawing of graphs on the plane [1 - 4].

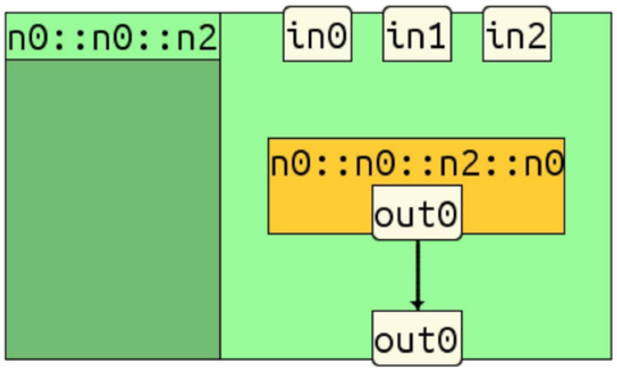

Fig. 3. A fragment n0::n0::n2 with tree arguments and one result which consists of single subfragment n0::n0::n2::n0.

## 3 General Description of Algorithm

Below it is assumed that the original IR-graph is a directed acyclic graph (DAG).

The algorithm consists in sequentially executing the steps of constructing images of the contents of fragments of the original graph, beginning with the innermost one, on each of which the drawing of a fragment is constructed with using of the sizes and locations of the elements of subfragments directly enclosed in it.

The construction of the image of a fragment is based on the technique of the so-called the hierarchical approach for creating layered drawings of DAGs, which was proposed by K. Sugiyama, and consists of the following three main stages [1], [2]:

(1) layer assignment when vertices to horizontal layers are assigned and thus their *y*-coordinates are determined,

(2) crossing reduction when orders of vertices within each layer to reduce the number of arc crossings are determined,

(3) horizontal coordinate assignment when an *x*-coordinates for each vertex is determined.

## 4 Layer Assignment

The task of this stage is to assign to each vertex its *y*-coordinate. For this, the source graph $G = (V, E)$ must be reduced to a layered digraph, which is a partition of $V$ into subsets $L_1, L_2, \dots, L_h$, such that if $(u,v) \in E$, where $u \in L_i$ and $v \in L_j$, then $i > j$. It is assumed that all vertices of the same level are assigned the same vertical *y*-coordinate, i.e. $v_y = i$, for all vertices from the level $L_i$. The height of a layered digraph is the number $h$, and the width $w_i$ is the number of vertices in the largest $L_i$. The span of arc $(u,v)$ with $u \in L_i$ and $v \in L_j$ is $i - j$. The layered digraph is said to be *proper* if no arc with a span greater than one.

The existence of a layered digraph for any DAG is obvious. However, not for every DAG there is a proper layered digraph (see Fig. 5, a).

To make a proper layered digraph, the technique of inserting “dummy vertices” can be used. Each arc $(u, v)$ of span $k=i–j>1$ is replaced with path $(u=v_1, \dots v_k=v)$ where $v_2, \dots v_{k-1}$ are added dummy vertices and $v_m \in L_{i-m+1}$ for all $m$ (see Fig. 5, b). The dummy vertices are needed because the crossing stage assumes that the digraph is proper.

There are three important requirements of the layering that are taken into account by this stage of our algorithm.

Firstly, the layered directed graph should be compact. This means that its height and width should be small.

Secondly, the layering should be proper. This is easily achieved by inserting “dummy vertices”.

Thirdly, the number of dummy vertices should be small.

So, the aim of the layer assignment stage is to transform into a layered digraph an acyclic directed graph $G = (V, E)$ which represents some fragment $F$ in such a way that $V$ consists of the ports of the fragment $F$ and the vertices directly contained in $F$.

Let $t$ be the length of the longest path in $G$. Then a layered digraph $L_1$, $L_2$, … , $L_{t+1}$ in which $L_1$ consists of all inputs of $F$, and $L_{t+1}$ consists of all outputs of $F$ will be constructed as follows.

First, $L_1$ and $L_{t+1}$ are constructed from the ports of the fragment and these ports are removed from $G$ together with the incident arcs. The process of construction of layered digraph continues in steps, on each of which one of the vertices that does not have incoming arcs in the current state of $G$ is included in the set $L_{i+1}$, where $i$ is the maximum number of the set $L_i$ containing its predecessor in the source graph $G$, with simultaneous removal of this vertex from $G$ together with all arcs outgoing from it. Moreover, among the vertices that do not have incoming arcs in the current state of $G$, the vertex that has the least number of incoming arcs and the largest number of outgoing arcs in the original graph $G$ (these numbers are pre-counted for all vertices of the original graph) is selected and included in the corresponding set, . This process continues until the current graph $G$ becomes empty. After this, the layered digraph $G$ is reduced to a proper one using the technique of dummy vertices.

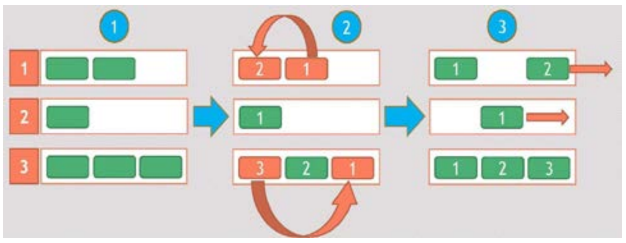

Fig. 4. The three main stage of the construction of the image of a fragment

## 5 Crossing Reduction

The task of this stage is to find the order of vertices at each level, in order to minimize the number of intersections of arcs.

It should be noted that the number of intersections of arcs in a layered digraph does not depend on the precise position of the vertices, but depends only on their relative position within each layer (their ordinal number at a given level). Thus, the task of this stage is not just a geometric problem, but merely a combinatorial one. However, this problem is NP-complete already for a graph having only two layers.

The execution of this stage for a given fragment is carried out as follows:

1. If a fragment has input and/or output ports located at $L_1$ and/or $L_h$, respectively, then the corresponding serial numbers should be assigned to these ports.

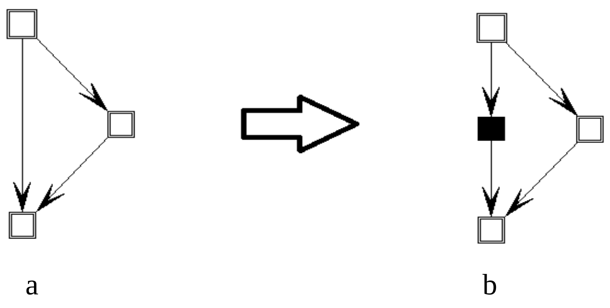

Fig. 5. Adding dummy vertex (drawn as dark square) to break up long edge in the layered digraph.

2. Consider the vertices of the $L_1$ level in their ordering, if the fragment has no ports, or $L_2$, if it has ports, and traverse the contents of the fragment starting from these vertices using the stack.

3. If the stack is empty, either we continue step 2, or this stage is completed. Otherwise, consider the vertex located at the top of the stack, but do not remove it from the stack. If there are no successor numbers for the vertex in question, then we assign the current order number to the vertex, delete it from the stack and go to step 3. If there are such successors at the top, we add one of them to the stack and go to step 3; when choosing a successor for placement on the stack, consider the following:

• If all successors of a given vertex are connected by arcs outgoing from different ports, then the order of inclusion of these vertices does not contradict the order of the ports.

• If there are vertices among successors associated with vertices that have already received sequence numbers, then they are included earlier than others.

• If there are fictitious peaks among the successors, they are selected in such order that they are located at the level in the middle, and the real tops on the level edges.

## 6 Horizontal Coordinate Assignment

After determining the order of the vertices at the level, it is necessary to determine their real coordinates. Arcs of the graph are represented in the form of broken lines with fracture points located in fictitious vertices; therefore, the task of determining the final coordinates of all vertices is simultaneously the task of carrying out arcs. If the arcs of the graph have some other form and / or method of conducting, then the corresponding criteria should be considered when solving the problem of determining the coordinates of the vertices.

At the input of this stage, we have a vertex division by levels $L_1$, $L_2$, … , $L_h$, the vertices on each level are of the order from 1 to $w_i$, where $i$ varies from 1 to $h$. And the largest number of vertices is at the level of $L_1$ (or at the level of $L_{h-1}$, if there are no output ports).

Beginning with the last level, using the barycenter method in combination with the vertex order constraints obtained in the previous step, determine the $x$-coordinate at the previous level. The vertices of the last level are distributed uniformly on a certain segment of the horizontal line allocated to the vertices of this level. Then, for each next level, the coordinates of its vertices are successively determined as the arithmetic mean of the coordinates of their neighbors from the already set levels. At the same time, the initial vertex order is not violated at the level.

If at some step the layering takes place, because the two vertices are assigned one $x$-coordinate, then the last level should be slightly expanded. For example, suppose that initially the vertices at this level have the $x$-coordinates 0, 1, 2, 3, 4, etc. At the next step they will have the $x$-coordinates 0, 2, 4, 6, 8, etc. And if you do not succeed in laying down the graph, you can still multiply the base: 0, 3, 6, 9, etc.

## 7 Implementation of the Algorithm

The described algorithm has been implemented within the Visual Graph system [9], [13].

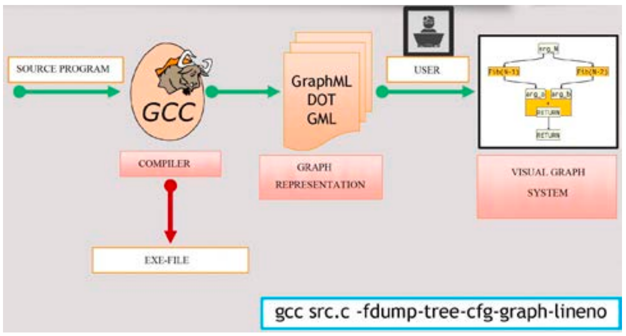

Fig. 6. Using the Visual Graph system by a user of the GNU Compiler Collection (GCC).

The Visual Graph system has been developed to visualize the internal data structures typically found in compilers and other programming systems. Data structures that occur in these systems are usually represented as attributed hierarchical graphs of big size. For example, the attributed syntax trees are used as the internal representations of translated programs in almost all compilers or interpreters. Optimizing and restructuring compilers require static analysis of control and data relationships in a program and their presentation in the form of a more general graph model of the program, for example, such as the control-flow graph.

It is assumed that the Visual Graph system is used as follows (Fig. 3). First, a compiler (or another programming system) itself or with an auxiliary program transforms a graph model arisen during compiling a source program from its internal representation into a graph representation (a file of one of the formats supported by the Visual Graph system, usually into the GraphML-file). Then the Visual Graph system will be able to read this graph model from the file, to visualize it and to provide a user with different navigation tools for its visual exploration.

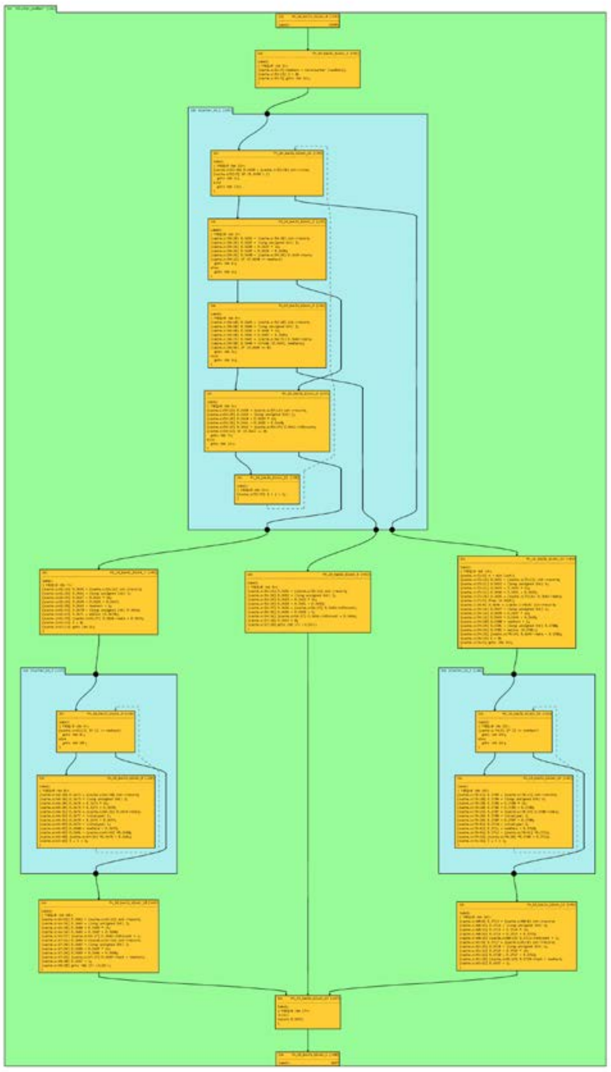

Fig. 7. An example of drawing of a control graph with loops. Bacwards arcs are shown by dashed lines.

In describing the algorithm, it was assumed that every IR-graph is a DAG. However, its implementation within the Visual Graph system was performed for more general graphs and includes also the steps of preliminary and final processing of the graph, which allow the system to work not only with DAGs (See Fig. 7).

The essence and purpose of these transformations lies in the reversible transformation of the structure of the original graph.

If the original graph contains undirected edges, then as a preliminary step it is realized its direction in accordance with the traversal of the graph in depth with the removal of orientation in the final processing.

If the original graph contains loops, then to obtain DAG the orientation of some of the arcs (so-called backwards arcs) of the loops is changed, and then the drawing of the original graph is constructed by the inverse transformation in the constructed drawing of DAG.

## 8 Conclusion

The problem of drawing of hierarchical graphs with ports has been considered and solved. An effective algorithm for visualization of IR-graphs and its effective implementation within the framework of Visual Graph system for visualization arbitrary attributed hierarchical graphs with ports are described.

The algorithm for visualization of IR-graphs has a quadratic time complexity and constructs a good drawing of any IR-graph. Its implementation within the framework of the Visual Graph system can be used for drawing of an arbitrary attribute hierarchical graph with ports and allows on an ordinary PC to obtain in real time (without visible delays) a good drawing of graph containing up to 10000 elements.

**Acknowledgments.** This research was supported by the Russian Science Foundation under grant RSF N 18-11-00118.

*References:*

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

[2] Sugiyama, K. Graph drawing and applications. For software and knowledge engineers, World Scientific, 2002.

[3] Herman I., Melancon G., Marshall M. S. Graph visualization and navigation in information visualization: a survey, *IEEE Trans. on Visualization and Computer Graphics*, Vol. 6, 2000, pp. 24–43.

[4] Kasyanov, V.N., Kasyanova, E.V., Information visualization on the base of graph models, *Scientific Visualization*, Vol. 6, No. 1, 2014, pp. 31–50. (In Russian).

[5] The aiSee system, http://www.aisee.com.

[6] The yEd system, http://www.yworks.com.

[7] The Cytoscape, http: // www. cytoscape. org.

[8] The Higres system, http://pco.iis.nsk.su/higres.

[9] Kasyanov, V., Zolotuhin, T. A system for structural information visualization based on attributed hierarchical graphs, *WSEAS Transactions on Computers*, Vol. 16, 2017, pp. 193–201.

[10] Kasyanov, V.N., Kasyanova, E.V., Methods and system of cloud parallel programming, in *Proceedings of the XIV International Asian School-Seminar on Problems of Optimizing Complex Systems*, Almaty, 2018, Part 1, pp. 298–307 (In Russian)

[11] Kasyanov, V.N., Kasyanova, E.V., *Programming language Cloud Sisal*, 2018. (In Russian)

[12] Kasyanov, V.N., Hierarchical graphs and graph models: problems of visual processing, in *Problems of Informatics Systems and Programming*, Novosibirsk, 1999, pp. 7-32. (In Russian)

[13] Kasyanov, V.N., Methods and tools for structural information visualization, *WSEAS Transactions on Systems*, Vol. 12, No. 7, 2013, pp. 349–359.

[14] Feng, Q.W., Cohen, R.F., Eades, P., Planarity for clustered graphs, *Lecture Notes in Computer Science*, vol. 979, 1995, pp. 213–226.

[15] Ershov, A.P., Theory of program schemata, In: *Proc IFIP Congress*, 1971, pp. 144–163.