Carassius: A Simple Process Model Editor

N. Nikitina <nmnikitina@edu.hse.ru>,
A. Mitsyuk <amitsyuk@hse.ru>,
PAIS laboratory, National Research University Higher School of Economics,
125319, Kochnovsky, 3, Moscow, Russia

Abstract. Process models of different types and graphs are commonly used for modeling and visualization of processes in information systems. They may represent sets of objects, tasks or events involved in process linked with each other in some way. Wide use of process models in various notations engenders necessity of software tools for creating, editing, and analysing them.

This paper describes the process model editor which allows for dealing with classical graphs, Petri nets, finite-state machines and systems of communicating automata. Additionally, the tool is armed with the following list of useful features: process simulation based on a Petri net token-based replay, import and export of process models in different persistent formats, various model layouts and other process visualization abilities. Moreover, Carassius is a modular tool which can be extended with additional process model notations, processing, import and export possibilities.

In the paper one can find a detailed description of a couple of layout algorithms already implemented in the tool. These algorithms for visualization of Petri nets and graphs can be used as a base point for further development of more refined process visualization approaches. Carassius might be useful for educational and research purposes because of its simplicity, range of features and variety of supported notations.

Keywords: graph; Petri net; finite-state machine; process model; process model visualization; process model editor

DOI: 10.15514/ISPRAS-2015-27(3)-15

For citation: Nikitina N., Mitsyuk A. Carassius: A Simple Process Model Editor. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 3, 2015, pp. 219-236. DOI: 10.15514/ISPRAS-2015-27(3)-15.

1. Introduction

The modern world is full of information systems working in different business domains. One of the most developed concepts is process-aware information systems [1]. A wide variety of different notations has been developed to model processes.

In this paper we present a new tool for editing and simulating process models in different notations. Our goal is not to build yet another complicated model simulator.

Our ambition was to develop a model editor which may be used for educational purposes. Thus, the decision was made to implement a simple and extensible model editor for different modeling notations. In particular, a modular architecture of *Carassius* allowed us to implement simulation modules in addition to different editors.

The remainder of this work is organized as follows. Section 2 gives a description of the tool, implemented approaches and algorithms. Furthermore, the description of the tool's features is provided.

In section 3 we consider other tools with similar functionality. The advantages and disadvantages of these tools are provided. Section 4 concludes the paper.

2. Tool Overview

2.1 Functionality

Here one can see the brief description of all features implemented in Carassius.

In this paper we present a tool which intended to help researchers and other people easily make and edit models of different types. Carassius works with graphs of 3 types: classical graphs, Petri nets and finite-state machines. First of all, it permits to edit process models by hand. Besides, the tool supports several markup languages (PNML [2], [3], GraphML [4], [5] and FSAML) and can read and save models from and into these formats. FSAML is a new XML format we developed for storing a finite state machines system.

The working area has a grid helping users position the nodes. The tool can automatically arrange model elements according to the grid. Users may set or change all the possible properties of the whole model or its parts (for example: node names, arc weights etc.) The tool can arrange models using different layout algorithms: for graphs and finite-state machines it uses the force-directed algorithm, whereas for Petri nets it uses the layering algorithm developed for Carassius. Both of them are described in details in subsection Visualization refinement.

In addition, Carassius has features for a Petri net simulation. The tool supports step-by-step token-game of a process model [6]. Moreover, there is a special coloring mode that shows the real way of tokens during the simulation. Because of these features, the tool can be used successfully in educational purposes.

2.2 Supported Notations

This section describes the modeling notations supported by Carassius.

2.2.1 Petri Nets

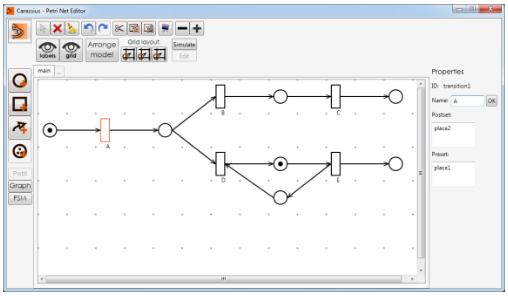


Fig.1. A Petri net editing.

The main supported formalism is Petri nets. Petri nets are widely used in process modeling [6], [7]. A Petri net is a directed bipartite graph with two types of nodes: transitions (denoted by rectangles) and places (denoted by circles). There are directed arcs between places and transitions (denoted by arrows). Places can contain so-called tokens inside, which determine the current state of a net and its marking. Petri nets offer a graphical notation for step-by-step processes that include choice, iteration, and concurrent execution. Execution of a process is depicted by tokens flow.

2.2.2 Graphs

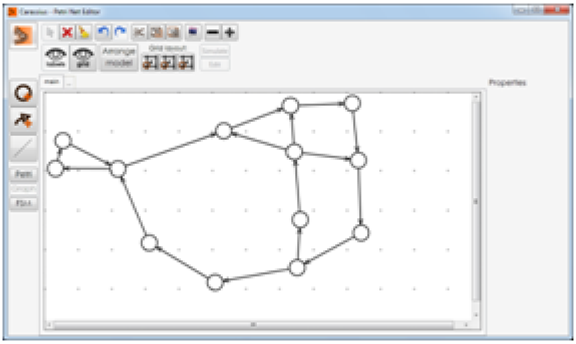


Fig.2. A graph editing.

Carassius is also works with classical graphs. Both directed and undirected edges are supported. It is possible to assign weights of edges. Process of graph editing is quite simple. However, a possibility to deal with directed graphs and store them using GraphML format is very useful.

2.2.3 Finite-State Machines

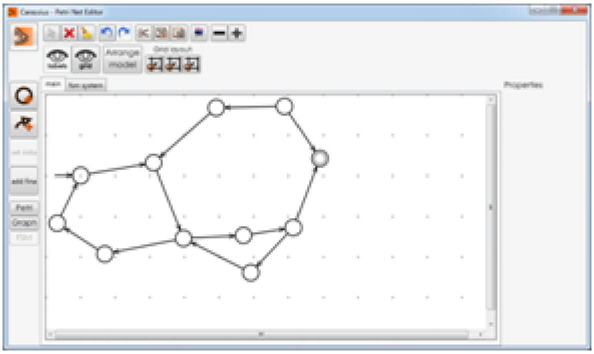


Fig.3. A finite-state machine editing.

A finite-state machine (FSM, finite-state automaton [8]) is an abstract machine that can be in an only one of a finite number of states at a point of time.

FSM recognizes or accepts certain word of some language with finite alphabet. It can move from one state to another by triggering a transition with the same label as a next letter of an input word. If a FSM stops in a state from the set of so-called acceptance states, then it accepts a word. This is not always the case. Therefore, any FSM forms a language consisting of the words accepted by this FSM.

A particular FSM is defined by a list of its states and transitions. States are usually depicted by circles, and transitions are depicted by labeled directed arcs. There are two special types of states: a single starting state and a set of final (accepting) states. A starting state is depicted by a circle with an arrow from anywhere going into the circle (see figure 3). Each accepting states is depicted by a double circle.

2.2.4 Systems of Finite-State Machines

Systems of communicating FSMs are also supported by Carassius. A system of Finite-State Machines may be useful for modeling processes which appear at the same time and have causal dependencies. A Finite-State Machine System deals with some number of FSMs and relations between them. These relations may be of two types: (1) synchronous (two transitions from the FSMs may fire only at the same time) and (2) asynchronous (there is a special state in-between the FMSs called the channel state). Synchronous relations are denoted by simple lines between two models, which hold the information about transitions which are fired simultaneously. Asynchronous - by sequence of arrow, place and another arrow, meaning that some action performed in one fsm may have consequences in another.

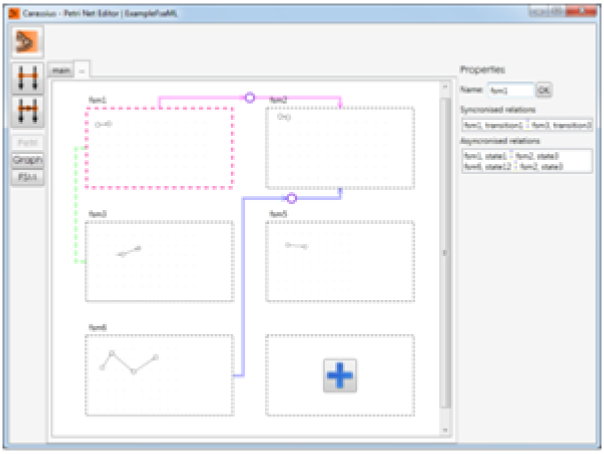


Fig.4. A system of finite-state machines editing.

2.2.5 Import and Export Formats

Carassius provides different import and export formats to facilitate work with models. It deals with several convenient markup language formats for import: PNML for Petri nets, GraphML for graphs, and FSAML for finite-state machines and their systems. All of them are XML-based interchange formats. In addition, one can easily export a model to png-picture or tikz-picture to import model to a TEX file.

2.2.5.1 Markup language formats

PNML and GraphML formats are well-known in the world of modeling and have been in use for a long time. Both of them have a clear specification and will be described further. On the contrary, FSAML (*Finite-State Automaton Markup Language*) has been developed recently by the authors of this paper and has not been formally described yet.

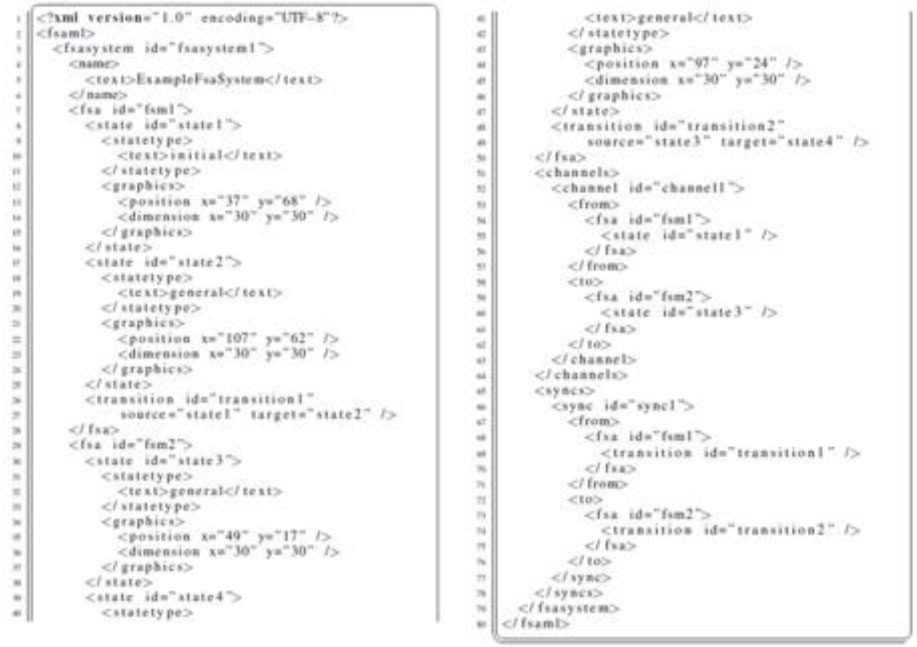
A detailed explanation of a PNML format can be found in [9]. A typical PNML file contains information about a net, a number of pages, lists of places, transitions and arcs. A lot of additional information is available such as names of nodes, dimensions etc. PNML is an extensible format. So, it is possible to make different extensions for particular modeling aspects. It is impossible to cover all extensions. That is why Carassius deals with PNML files according to the recent version of the core standard (ISO/IEC 15909-2:2011).

GraphML is a comprehensive and easy-to-use file format for graphs. It consists of a language core for describing the structural properties of a graph. A detailed description can be found in [10]. Carassius, in turn, supports only simple graphs (directed, undirected and mixed) without any additional features.

FSAML is a format allowing exchange of finite-state machines and their systems. The development of this format is still in progress. However, there is a working alpha implementation of it in Carassius.

The structure of the file according to the format is following: the main node (*fsasystem*) consists of its name (*name*), a number of finite-state machines (*fsa*), synchronous (*syncs*) and asynchronous (*channels*) relations between them. In turn, a *fsa* node contains a number of states (*state*) and transitions (*transition*). Each of them has an attribute id holding unique id. Each state has its type: general, initial or final, therefore there is an inner node *statetype* containing this information. The second inner node is *graphics* representing the data about position and dimension of a node. Transitions have their source states (*source*) and target states (*target*) represented as attributes. The *channels* node consists of several channels (*channel*), which, in turn, have two nodes: *from* and *to* containing information about *fsa* and a corresponding *state*. The *syncs* node has the same structure except the fact that relation is between two transitions, not states.

An example of the file in the FSAML format is shown on figure 5.



```
<?xml version="1.0" encoding="UTF-8"?>
<fsaml>
  <fsasystem id="fsasystem1">
    <name>
      <text>ExampleFsaSystem</text>
    </name>
    <fsa id="fsm1">
      <state id="state1">
        <statetype>
          <text>initial</text>
        </statetype>
        <graphics>
          <position x="37" y="68" />
          <dimension x="30" y="30" />
        </graphics>
      </state>
      <state id="state2">
        <statetype>
          <text>general</text>
        </statetype>
        <graphics>
          <position x="107" y="62" />
          <dimension x="30" y="30" />
        </graphics>
      </state>
      <transition id="transition1"
          source="state1" target="state2" />
    </fsa>
    <fsa id="fsm2">
      <state id="state3">
        <statetype>
          <text>general</text>
        </statetype>
        <graphics>
          <position x="49" y="17" />
          <dimension x="30" y="30" />
        </graphics>
      </state>
      <state id="state4">
        <statetype>
          <text>general</text>
        </statetype>
        <graphics>
          <position x="97" y="24" />
          <dimension x="30" y="30" />
        </graphics>
      </state>
      <transition id="transition2"
          source="state3" target="state4" />
    </fsa>
    <channels>
      <channel id="channel1">
        <from>
          <fsa id="fsm1">
            <state id="state1" />
          </fsa>
        </from>
        <to>
          <fsa id="fsm2">
            <state id="state3" />
          </fsa>
        </to>
      </channel>
    </channels>
    <syncs>
      <sync id="sync1">
        <from>
          <fsa id="fsm1">
            <transition id="transition1" />
          </fsa>
        </from>
        <to>
          <fsa id="fsm2">
            <transition id="transition2" />
          </fsa>
        </to>
      </sync>
    </syncs>
  </fsasystem>
</fsaml>
```

Fig.5. The FSAML format.

2.2.5.2 TEX and PNG export

The tool has features for TEX and PNG export. Carassius may generate a code to import picture using tikz-package into your TEX file. Figure 6 shows a simple Petri net edited with Carassius and exported directly into TEX. This feature has been implemented with help of N. Chuykin (a student at HSE).

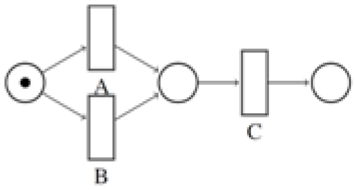


Fig.6. A picture compiled with tikz package.

2.3 Visualization refinement

The presented tool has several features to make model visualization better. There are two special algorithms for the directed graphs and for Petri nets, which can arrange nodes to make model easier to understand. Graphs and Petri nets can be processed in different ways. The tool also provides a grid for working area which helps placing nodes more accurately. Finally, Carassius provides possibility to hide/show grid as well as node labels. This section describes the layout algorithms in detail.

2.3.1 Petri Net layout

Firstly, the layout refinement algorithm for Petri nets is described. It is a layered-based algorithm which was developed especially for Petri nets. Layered-based algorithms are a group of layout algorithms which work with directed graphs and take their hierarchical structure into account [11]. We chose this approach as the most suitable for Petri nets as they are directed, and bipartite. The structure of the Petri nets notation is quite suitable for a layered representation. The main scheme of the layered-based approach is described in [12]. These algorithms are aimed to cover the list of aesthetic points:

1) single edges direction,
2) occupied area minimization,
3) uniform nodes allocation,
4) long edges avoidance,
5) edges-crossing minimization.

Although some of these points may conflict with each other, the approach is viable. It works using three steps:

1) allocation of nodes on layers in a way which ensures that edges have single direction;
2) choice of the nodes order on layers with the aim of edges-crossing minimization;
3) determination of node coordinates on layers with the aim of edges-length minimization.

In the presented algorithm these three ideas are used, but some features are added and changed as well.

The algorithm in Carassius takes into account: (1) a biparticity of Petri nets, (2) the fact that they have directed arcs, and (3) a presence of initial places.

```
Data: List of all nodes as nodes
Result: All nodes are arranged
int modelNumber = 1;
while each node doesn't belong to any model do
    Node firstNode = findNodeWithoutModelNumber();
    depthFirstSearch(firstNode, modelNumber);
    modelNumber++;
end
foreach model do
    List<Node> modelNodes =
    getAllModelNodes(modelNumber);
    List<Node> initialNodes =
    searchForInitialNodes(modelNodes);
    setColumnForStartingNodes(startingNodes);
    setColumnForEachNode(modelNodes);
    setYcoordinateForEachNode(modelNodes);
    setSpaceBetweenColumns();
end
visualize();
return coordY
```

Alg.1. Petri net layout algorithm.

Generally, it determines connected components of a model (a number of individual graphs in one model), applies layered-based approach for each component and then gathers components together to visualize an overall model. We use so-called 'columns' to represent layers. Due to the Petri nets biparticity the content of columns alternates from places to transitions. We start from the first column with places. When several steps of the algorithm are made, each node has its column (using breadth-first search), and we can arrange nodes in each column separately (set them y-coordinate). The overall algorithm 1 shows all the steps.

```
Data: Initial node as node, number of model as modelNum
Result: All nodes of the model are marked
foreach Arc arc in node.thisArcs do
    Node next;
    if arc.To == node then
        next = arc.To;
    else
        next = arc.From;
    end
    next.modelNumber = modelNum;
    next.isChecked = true;
    foreach Arc arc1 in next.thisArcs do
        Node next1;
        if arc1.To == node then
            next1 = arc1.To;
        else
            next1 = arc1.From;
        end
        if next1.isChecked == false then
            next1.isChecked = true;
            next1.modelNumber = modelNum;
            depthFirstSearch(next1, modelNum);
        end
    end
end
```

Alg.2. Determination of all nodes in a model.

```
Data: List of all nodes as nodes
Result: List of initial nodes as initialNodes
List<Node> initialNodes = new List<Node>(); foreach Node
node in nodes do
    if node.thisArcs.Count == 0 then
        initialNodes.Add(node);
    else
        bool hasIngoingArcs = false;
        foreach Arc arc in node.thisArcs do
            if arc.To == node then
                hasIngoingArcs = true;
                break;
            end
        end
        if hasIngoingArcs == false then
            initialNode.Add(node);
        end
    end
end
return initialNodes
```

Alg.3. Search of initial nodes.

In order to arrange nodes the tool makes the following steps:

(a) Determines connected components of the models. A Petri net model may consist of several individual connected components, so we have to detect them. Also, for each set of nodes we have to assign the number used for component identification.

Next steps are done for each connected component of the model:

(b) Finds all initial nodes (both transitions and places). A node considers as initial if it doesn't have any ingoing arcs.

(c) Sets columns for the initial nodes. This step is needed because these nodes will become starting points to move through the graph.

(d) Sets a column for each node. This algorithm is layered-based, thus, we need to distribute nodes among columns.

(e) Sets a y-coordinate for each node. At this step we want to place each node in some place at a column. To make the model layout more compact we locate nodes symmetrically from the center of a column (mean value between minimal and maximal y-coordinate of nodes in a column).

(f) Sets margin between columns. There may be very few or, on the contrary, too many arcs between the nodes in two adjacent columns. So, these distances should depend on a number of arcs between neighbor columns.

(g) Visualizes the whole model. The whole model is visualized using all information derived at the previous steps.

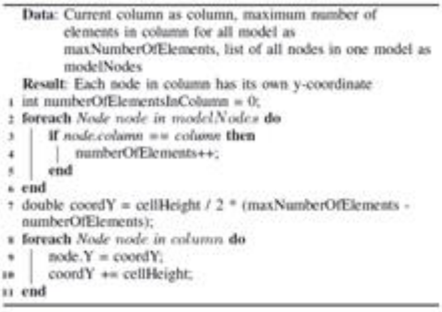
The listing 2 shows the algorithm which divides a model into several connected components. To obtain the list of initial nodes the algorithm 3 is used.

```
Data: List of all nodes as nodes
Result: Each node has its column
int currentColumn = 1;
while each node hasn't its column do
    List<Node> currentColumnNodes = new List<Node>();
    foreach Node node in nodes do
        if node.column == currentColumn then
            currentColumnNodes.Add(node);
        end
    end
    foreach Arc arc in node.thisArcs do
        Node temp;
        if arc.To == node then
            temp = arc.From;
        else
            temp = arc.To;
        end
        if node.column == 0 then
            node.column = currentColumn + 1;
        end
    end
    currentColumn++;
end
```

Alg.4. Search of a column for each node.

The distribution of all nodes in columns is shown in the algorithm 4.

Algorithm 5 arranges each node for its place (y-coordinate) in a column.



```
Data: Current column as column, maximum number of
      elements in column for all model as
      maxNumberOfElements, list of all nodes in one model as
      modelNodes
Result: Each node in column has its own y-coordinate
int numberOfElementsInColumn = 0;
foreach Node node in modelNodes do
    if node.column == column then
        numberOfElements++;
    end
end
double coordY = cellHeight / 2 * (maxNumberOfElements -
numberOfElements);
foreach Node node in column do
    node.Y = coordY;
    coordY += cellHeight;
end
```

Alg.5. Setting of a position for each node in a column.

2.3.2 Graph layout

In this subsection the layout algorithm for graphs is described. Carassius contains implementation of the existing algorithm from [13] with little changes. It is a force-directed algorithm aspired to achieve several goals:

(1) nodes should not be too close to each other,

(2) edges should have more or less equal length and do not cross each other too often.

This algorithm does a number of iterations to achieve the best arrangement of a graph. It is done by assigning so-called forces and velocities among the set of edges and the set of nodes, based on their relative positions.

```
Data: List of all nodes in one model as nodes, list of all arcs in
      one model as arcs
Result: All nodes in one model are arranged
double oldX, oldY, newX, newY;
foreach Node node in nodes do
    // nextDouble returns a real number from 0 to 1 node.X =
    200 + nextDouble() * 300;
    node.Y = 100 + nextDouble() * 200;
end
do
    for i ← 0 to nodes.Count do
        nodes[i].netForceX = nodes[i].netForce.Y = 0;
        for j ← 0 to nodes.Count do
            if i == j then
                continue;
            end
            double squaredDistance =
            $(node[i].X - node[j].X)^2$ +
            $(node[i].Y - node[j].Y)^2$;
            nodes[i].netForceX += 200 * (nodes[i].X -
            nodes[j].X) / squaredDistance;
            nodes[i].netForceY += 200 * (nodes[i].Y -
            nodes[j].Y) / squaredDistance;
        end
        foreach Arc arc in arcs do
            Node tempNode;
            if arc.From == nodes[i] then
                tempNode = arc.To;
            else
                tempNode = arc.From;
            end
            nodes[i].netForceX += 0.06 * (tempNode.X -
            nodes[i].X);
            nodes[i].netForceY += 0.06 * (tempNode.Y -
            nodes[i].Y);
        end
        nodes[i].velocityX = (nodes[i].velocityX +
        nodes[i].netForceX) * 0.85;
        nodes[i].velocityY = (nodes[i].velocityY +
        nodes[i].netForceY) * 0.85;
    end
    oldX = nodes[0].X;
    oldY = nodes[0].Y;
    foreach Node node in nodes do
        node.X += node.velocityX;
        node.Y += node.velocityY;
    end
    newX = nodes[0].X;
    newY = nodes[0].Y;
while oldX != newX || oldY != newY;
```

Alg.6. Force-based algorithm for a graph model layout.

An algorithm for graph layout in Carassius consists of two main steps:

(a) The force-directed algorithm (see algorithm 6) itself. It is applied for each connected component. Constants used in the algorithm were selected experimentally based on application UI configuration.

(b) A movement of all nodes on fixed distances. Nodes can have negative coordinates after applying the algorithm, so we need to move them because working area shows only those which have positive coordinates. We also need to do some movements to place models in such a way in order to save a distance between them.

2.4 Simulation

Petri nets are not only simple bipartite graphs but also a powerful tool able to represent a process flow. There are ‘tokens’ (markers inside places), reflecting current state of a net. They can change their places by the transitions firing. A transition may be fired if all places which have outgoing arcs to this transition have enough tokens inside (equal or more than weight of a corresponding arc). At each step only one transition is fired (may be chosen by hand or randomly). When a transition is fired it consumes the required number of tokens and passes a token to each outgoing place. The simulation ends when there is no transition able to be fired.

Simulation of an example Petri net made in Carassius is shown in figure 7.

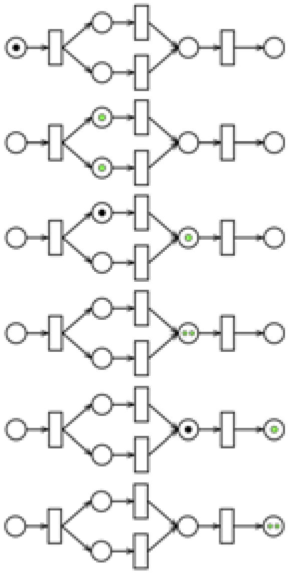


Fig.7. Simulation of a Petri net.

2.4.1 Wave coloring

Simulation of a net in our tool may also be done in a waving mode. During simulation nodes are colored in a specific way. A movement of a token from one place to another will be considered as a single step. Nodes engaged in the last step have deep blue color, whereas nodes used in previous steps are colored in light blue. In other words, the later a step is made, the darker a node is colored, the earlier – the lighter. This coloring allows for easily understanding of a process direction, determining which nodes were visited and which were not.

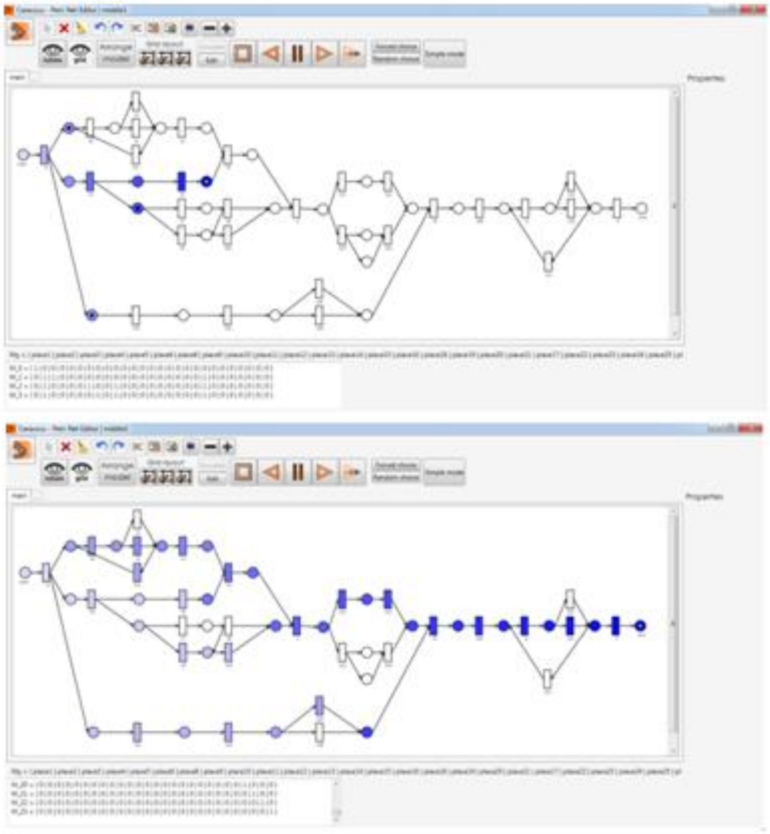


Fig.8. Wave coloring during simulation of a Petri net.

Figure 8 shows how wave coloring of a simulation works in Carassius. The top part of the picture shows simulation at the intermediate step. The bottom part shows a window when the simulation has been ended.

2.5 Architecture

The tool is built as a standalone windows application using C#. We used the Windows Presentation Foundation (WPF) platform to build our application because of its functionality, extensibility and convenience. The WPF provides user controls as a mechanism for reusing blocks of the UI elements. The main window of Carassius consists only of one user control, which may be easily moved to another application as a component.

3. Related work

A variety of model editors are available now. Nevertheless, all of them did not fully meet our two main requirements (simplicity and extensibility). This section describes the closest existing tools which support model editing in a desirable way.

a) *CPN Tools* (see [14]): CPN Tools is a tool for working with Colored Petri nets. It allows users to edit, simulate, and analyze them. CPN Tools has an interesting, original interface which uses a lot of small inner windows for each type of editing. However, at first a user can get stuck because the GUI is not very intuitive and the user needs to read the help to understand what he should do in order to start working. In addition, the tool works only with colored Petri nets and you cannot work with simple ones.

b) *Yasper* (see [15]): Yasper, as authors say, is the yet another smart process editor. It is a quite simple, but useful tool which supports editing and simulation of Petri nets. It has rather user-friendly and easy to use interface, but it is still unevident how to do some actions. Fortunately, its help paper is very useful and provides a lot of information about usage of the tool. However, Yasper has a significant drawback - it does not support the current version of the PNML format, so the user just cannot download new PNML files and cannot work with exported files from the tool anywhere else.

c) *Tina* (see [16]): Tina is a tool for working with classical P/T and Time Petri nets. It has features for editing and analysis of Petri nets. Tina's interface is very simple, but at the same time easy to understand. Editing functionality is not very wide, but the tool provides several analysis techniques, which work well. Tina's disadvantage is that it cannot simulate Petri nets in a visual way and has a small number of functions.

We can see that several tools for working with Petri nets are already exist, but all of them have certain drawbacks. In our tool we endeavored to take into account all disadvantages we found in other tools, and at the same time to add new functionality. We tried to do interface easy to use and learnable, intuitive to work; to provide support of different export and import formats; to implement all main tasks which can be done with Petri nets; and, finally, to incorporate some new features (e.g. several visualization refinement algorithms).

4. Conclusion

A lot of features and several modes are already implemented in Carassius. One can use it to deal with graphs, Petri nets, Finite-State Machines. Due to modularity of the tool we want to extend it with other modeling formalisms. The most difficult thing is to preserve the simplicity of the software while adding new features.

Our tool has been used in different other projects at PAIS Lab [17], [18]. We hope it will also be useful for other researchers (see [19]).

Of course, there is still a lot of work to do. Our main goal is to improve the FSM aspect of the tool. This functionality is involved in other projects of our group.

Complete definition of the FSAML format is the key point of the future work. Moreover, we intend to add simulation functionality for the finite-state machines.

Another aim is to carry out a number of user tests in order to find and eliminate bugs in the tool. In addition, we are going to do usability testing to make Carassius more intuitive to use and work with. There are several possible improvements of GUI we want to implement.

Acknowledgment

We would like to thank members of the PAIS Lab for their support. Research assistants I. Shugurov and A. Begicheva tested the tool and reported lots of bugs. Dr. A. A. Kalenkova and prof. I. A. Lomazova gave us a valuable advice on the GUI design and the required features.

Also we would like to thank Nikolay Chuikin, who implemented the TEX-export used in the tool.

This work is output of a research project implemented as part of the Basic Research Program at the National Research University Higher School of Economics (HSE).

References

[1]. M. Dumas, W. M. van der Aalst, and A. H. ter Hofstede, Process-aware Information Systems: Bridging People and Software Through Process Technology. New York, NY, USA: John Wiley & Sons, Inc., 2005.

[2]. M. Weber and E. Kindler, “The petri net markup language,” in Petri Net Technology for Communication-Based Systems - Advances in Petri Nets, 2003, pp. 124–144.

[3]. J. Billington, S. Christensen, K. M. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber, “The petri net markup language: Concepts, technology, and tools,” in Applications and Theory of Petri Nets 2003, 24th International Conference, ICATPN 2003, Eindhoven, The Netherlands, June 23-27, 2003, Proceedings, 2003, pp. 483–505.

[4]. U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. Marshall, “Graphml progress report structural layer proposal,” in Graph Drawing, ser. Lecture Notes in Computer Science, P. Mutzel, M. Jnger, and S. Leipert, Eds. Springer Berlin Heidelberg, 2002, vol. 2265, pp. 501–512.

[5]. U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall, “Graphml progress report,” in Graph Drawing, 2001, pp. 501–512.

[6]. W. Reisig, Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies. Springer, 2013.

[7]. T. Murata, “Petri nets: Properties, analysis and applications,” Proceedings of the IEEE, vol. 77, no. 4, pp. 541–580, 1989.

[8]. J. A. Anderson, Automata theory with modern applications. Cambridge University Press, 2006.

[9]. L. Hillah, E. Kindler, F. Kordon, L. Petrucci, and N. Treves, "A primer on the petri net markup language and iso/iec 15909-2," Petri Net Newsletter, vol. 76, pp. 9–28, 2009.
[10]. U. Brandes, M. Eiglsperger, and J. Lerner, "Graphml primer," Online: http://graphml. graphdrawing. org/primer/graphml-primer. html [29.05.2007], 2004.
[11]. G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis, Graph Drawing: Algorithms for the Visualization of Graphs. Prentice-Hall, 1999.
[12]. V. Kasianov and V. Evstigneev, Grafi v programmirovanii. BHV - Peterburg, 2003. (In Russian)
[13]. S. G. Kobourov, "Spring embedders and force directed graph drawing algorithms," arXiv preprint arXiv:1201.3011, 2012.
[14]. M. Westergaard and L. M. Kristensen, "The access/cpn framework: A tool for interacting with the cpn tools simulator," in Applications and Theory of Petri Nets. Springer, 2009, pp. 313–322.
[15]. K. van Hee, O. Oanea, R. Post, L. Somers, and J. M. van der Werf, "Yasper: a tool for workflow modeling and analysis," in Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on. IEEE, 2006, pp. 279–282.
[16]. B. Berthomieu*, P.-O. Ribet, and F. Vernadat, "The tool tina – construction of abstract state spaces for petri nets and time petri nets," International Journal of Production Research, vol. 42, no. 14, pp. 2741–2756, 2004.
[17]. A. K. Begicheva and I. A. Lomazova, "Checking conformance of high-level business process models to event logs," in Proceedings of the Spring/Summer Young Researchers Colloquium on Software Engineering, vol. 8, 2014.
[18]. A. A. Mitsyuk and I. S. Shugurov, "On process model synthesis based on event logs with noise," Modeling and analysis of information systems, vol. 4(21), pp. 181–198, 2014.
[19]. N. Nikitina and A.Mitsyuk, "Carassius: A Simple Petri Net Editor," accessed: 2015-04-01. [Online]. Available: www.pais.hse.ru/research/projects/carassius

Редактор моделей процессов «Carassius»

Н. Никитина <nmnikitina@edu.hse.ru>,
А. Мицюк <amitsyuk@hse.ru>,
НУЛ ПОИС, Национальный Исследовательский Университет Высшая Школа Экономики, 125319, Россия, г. Москва, пр. Кочновский, д. 3.

Аннотация. Модели и графы процессов различных типов широко используются для моделирования и визуализации процессов в информационных системах. Такие модели представляют взаимосвязи между объектами, задачами или событиями в рамках процесса. Использование большого количества моделей процессов в разнообразных нотациях вызывает необходимость разрабатывать программные инструменты, обеспечивающие конструирование, редактирование и анализ моделей процессов.

Данная работа описывает инструмент для редактирования моделей процессов, обладающий функциями для работы с моделями в виде классических графов, сетей Петри, конечных автоматов и систем взаимодействующих конечных автоматов. Кроме этого, программа имеет следующий набор полезных функций: симуляция процессов на базе исполнения сетей Петри с использованием токенов, импорт и экспорт моделей процессов в различных форматах хранения, разнообразные способы автоматического графического размещения моделей на плоскости, алгоритмы визуализации процессов. Более того, модульная архитектура Carassius позволяет расширять инструмент, добавляя поддержку дополнительных нотаций моделей процессов, алгоритмов обработки и визуализации моделей, их импорта и экспорта. В данной статье предлагаются два алгоритма графического размещения сетей Петри и графов на плоскости, приводится описание их реализации в программном обеспечении Carassius. Эти алгоритмы могут служить основой для разработки других, более совершенных алгоритмов визуализации разных аспектов процессов.

В ходе проектирования и разработки инструмента Carassius особое внимание уделялось обеспечению простоты использования, внутреннего устройства и расширяемости. Благодаря этому представленное программное обеспечение может использоваться в образовательных и исследовательских целях.

Ключевые слова: граф; сеть Петри; конечный автомат; модель процесса; визуализация моделей процессов; редактор моделей процессов

DOI: 10.15514/ISPRAS-2015-27(3)-15

Для цитирования: Никитина Н., Мицюк А. Редактор моделей процессов «Carassius». Труды ИСП РАН, том 27, вып. 3, 2015 г., стр. 219-236 (на английском языке). DOI: 10.15514/ISPRAS-2015-27(3)-15.

Список литературы

[1]. M. Dumas, W. M. van der Aalst, and A. H. ter Hofstede, Process-aware Information Systems: Bridging People and Software Through Process Technology. New York, NY, USA: John Wiley & Sons, Inc., 2005.

[2]. M. Weber and E. Kindler, “The petri net markup language,” in Petri Net Technology for Communication-Based Systems - Advances in Petri Nets, 2003, pp. 124–144.

[3]. J. Billington, S. Christensen, K. M. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber, “The petri net markup language: Concepts, technology, and tools,” in Applications and Theory of Petri Nets 2003, 24th International Conference, ICATPN 2003, Eindhoven, The Netherlands, June 23-27, 2003, Proceedings, 2003, pp. 483–505.

[4]. U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. Marshall, “Graphml progress report structural layer proposal,” in Graph Drawing, ser. Lecture Notes in Computer Science, P. Mutzel, M. Jnger, and S. Leipert, Eds. Springer Berlin Heidelberg, 2002, vol. 2265, pp. 501–512.

[5]. U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall, “Graphml progress report,” in Graph Drawing, 2001, pp. 501–512.

[6]. W. Reisig, Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies. Springer, 2013.

[7]. T. Murata, “Petri nets: Properties, analysis and applications,” Proceedings of the IEEE, vol. 77, no. 4, pp. 541–580, 1989.

[8]. J. A. Anderson, Automata theory with modern applications. Cambridge University Press, 2006.

[9]. L. Hillah, E. Kindler, F. Kordon, L. Petrucci, and N. Treves, “A primer on the petri net markup language and iso/iec 15909-2,” Petri Net Newsletter, vol. 76, pp. 9–28, 2009.

[10]. U. Brandes, M. Eiglsperger, and J. Lerner, “Graphml primer,” Online: http://graphml. graphdrawing. org/primer/graphml-primer. html [29.05.2007], 2004.

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

[12]. Касьянов В. Н., Евстигнеев В. А. Графы в программировании //Обработка, визуализация и применение. БХВ-Петербург. – 2003.

[13]. S. G. Kobourov, “Spring embedders and force directed graph drawing algorithms,” arXiv preprint arXiv:1201.3011, 2012.

[14]. M. Westergaard and L. M. Kristensen, “The access/cpn framework: A tool for interacting with the cpn tools simulator,” in Applications and Theory of Petri Nets. Springer, 2009, pp. 313–322.

[15]. K. van Hee, O. Oanea, R. Post, L. Somers, and J. M. van der Werf, “Yasper: a tool for workflow modeling and analysis,” in Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on. IEEE, 2006, pp. 279–282.

[16]. B. Berthomieu*, P.-O. Ribet, and F. Vernadat, “The tool tina – construction of abstract state spaces for petri nets and time petri nets,” International Journal of Production Research, vol. 42, no. 14, pp. 2741–2756, 2004.

[17]. A. K. Begicheva and I. A. Lomazova, “Checking conformance of high-level business process models to event logs,” in Proceedings of the Spring/Summer Young Researchers Colloquium on Software Engineering, vol. 8, 2014.

[18]. A. A. Mitsyuk and I. S. Shugurov, “On process model synthesis based on event logs with noise,” Modeling and analysis of information systems, vol. 4(21), pp. 181–198, 2014.

[19]. N. Nikitina and A.Mitsyuk, “Carassius: A Simple Petri Net Editor,” accessed: 2015-04-01. [Online]. Available: www.pais.hse.ru/research/projects/carassius