CONNECTIVITY CONTROL IN AD HOC SYSTEMS: A GRAPH GRAMMAR APPROACH

Alexander Mikov, Alexander Borisov

Abstract: *We discuss the problem of connectivity within large-scale dynamic distributed information systems. Ad hoc system can adapt themselves through resource management or reconfiguration to achieve specific goals, such as functions, performance, energy budget and reliability. One of the mostly important goals is to keep a possibility of routing for any two nodes of the distributed system. Structure of the system can be described by a time-graph. In some moments of the time the structure of a network changes: new nodes can be included into the network and old nodes can be deleted. Thus the connectivity property of the time-graph is a logical function of time. A cause of disconnection may be technical (hardware failures, low energy, server overloading etc) or organizational (information secure, regular breaks or random interruptions). To improve the quality of service we have to control the distributed information system structure. The first step of a self-management (autonomic) system design is to describe permissible structures and forbidden structures. We propose to use the well-known method of graph grammars for this purpose. A finite set of graph grammar rules defines an infinite (but countable) set of permissible structures. An inference process allows us to get some permissible graph after doing of a finite sequence of steps. In this work we solve the problem of a set of graph grammar rules description for such property of graphs as connectivity. A detailed description of the rule set is used for rewriting of any graph (connected or disconnected) to a connected graph. Also we discuss a second step of a self-aware system design: including of the graph grammar into a feedback control cycle. An autonomic system has the property of self-awareness, i.e. the system contains its model and manages itself using this model. During a life-time cycle the current model can be compared with the permissible set described by graph grammar. If the current model isn't belong to the permissible set then the distributed system turns a connectivity renewal process on. Some problems of graph grammar algorithms complexity are discussed.*

Keywords: *time-graph, grammar, autonomic system, ad hoc system.*

ACM Classification Keywords: *H. Information Systems: H.3 Information Storage and Retrieval: H.3.4 Systems and Software – Distributed systems. C. Computer Systems Organization: C.2 Computer-Communication Networks: C.2.4 Distributed Systems – Distributed applications*.

Introduction

Autonomic computing is assumed solving problems (tasks) in self-managed computer environments. Such computer environment is based on local or global computer network, which functions almost without interference of the person – administrator of the network. Autonomic system is self-awareness, i.e. it contains its model and manages itself using this model. Frequently it is ad hoc network consisting in mobile nodes and which is created undertime (rather quickly) for a short time, when emergency situation is happened and so on. In this case self-management is very important. It can include self-generation, self-supporting in active functioned period and self-conservation at the completion of the network.

In this work some aspects of autonomic network structure modeling for the purpose of building software, which implements the model, for supporting self-managing algorithms, are under study.

Mobile networks graphs

At the generation of the network (time $t = 0$) specified links (relations) structure between computational nodes, which is defined by solving task (for example, placement of nodes and signal distance wireless connection) appears. It is natural that such structure can be represented by the graph $G(0) = G_0$. The graph $G(t)$ is changing in the process of functioning. In different times some nodes leave the system, but other nodes are included in it. Moving of nodes leads to increasing/decreasing distance between nodes and power of signal and respectively appearing new edge in graph $G(t)$ or removing existing edge [1]. Gradual reducing of battery charge also leads to decreasing of amount of edge in respective graph.

Commonly it is desirable (often necessary), that graph $G(t)$ has such property as permanent connectivity. Any pair of vertexes must be connected with at least one simple chain. The length of such chain is also important, because the quantity of hops has direct impact on latency of information transmission, i.e. qualitative characteristic of computer network. A routing task consists in searching of the shortest paths. Due to changing of graph $G(t)$ the routing task must be solved again after each change of the graph. Of course it can't be solved completely at the periods of disconnected graph [2].

So two conclusions can be made:

- not all graphs are acceptable and acceptable graphs have different qualitative characteristics;
- network must possess knowledge of its graph.

Networks, which belong to studied class, as a rule, are decentralized, that's why the question about placement of “knowledge” about graph is urgent. On the local level it is knowledge about neighbors (one-hop). Acquisition of local knowledge require the least cost, but it is difficult make a global decision (for example, routing) basing on local knowledge. The next level is knowledge about neighbors of neighbors (two hop), which is more difficult to receive and it is less reliable, because it quickly goes out of date. In general, the more knowledge is global the less it is reliable.

Graph acceptable is defined not only by connectivity, but also other characteristics of solving by the network task.

Label Ω as set of graphs. Graph $G(t)$ takes on a value from this set, $G(t) \in \Omega$, making in this set a trajectory. Based on the above the set Ω can be separated on two subsets: Ω_p – permissible and Ω_p – forbidden graphs of the networks [3].

Mobile networks grammars

The natural approach to defining sets of graphs with specified properties is graph grammars or graph rewriting systems. Grammar can be used for checking belonging of graph to the set $G(t) \in \Omega_p$, i.e. for solving the task of recognition.

Some approaches to graph rewriting systems exist. One of them is algebraic approach which based on the category theory. At the same time algebraic approach is divided to some approaches. Double-pushout approach (DPO) and single-pushout approach (SPO) are the most important.

Boolean algebra and matrix theory based approaches also exist and it is called matrix graph grammars.

Except of algebraic approaches, determine graph rewriting systems, which based on logic and database theory, can be separated out.

In this work algebraic approach – single-pushout approach (hereinafter referred to as SPO) – is studied.

In this approach rule *p*: ($L \rightarrow_r R$) consists of a rule name *p* and of an injective partial morphism *r*, called the rule morphism. Graphs *L* and *R* are called left-hand and right-hand side of the rule, respectively. Graph grammar *GG* is a pair $GG = \langle (p: r)_{p\in P}, S\rangle$, where $(p: r)_{p\in P}$ is a family of rule morphisms indexed by rule name, and *S* is the start graph of the grammar [4].

In practice classic graph grammar theory is not enough often. One of the additional mechanisms is application conditions. Application conditions allow setting the requiring context for application of the grammar rule. Application conditions separated to two classes: positive and negative application conditions. Positive constraints require existence of the elements (edges or vertexes) and vice versa, negative forbid the existence of some edges or vertexes [5].

Now describe graphical layout for representing application conditions. All constraints are distinguished by dotted border. If the area inside the dotted border is crossed by the line, it means that it is negative application condition else positive [6].

The next set of directives was developed to describe graph grammar in autonomic computer system software and also transform (edit) grammar:

– Directive CreateRule <identifier>.

– Directive

  Set <rule identifier> (*left* | *right*) *graph <graph identifier>*

  sets left or right graph in the rule depending on selected parameters (*left* or *right*).

– Directive *AddRule <rule identifier>* adds the rule to the grammar.

– Directive *RemoveRule <rule identifier>* removes rule.

– Directive

  AddConnectingRule <rule identifier>
  (*if* (*exist* | *absent*) *arc <vertex identifier >* (*in* | *out*) *<vertex identifier>*
  then make arc <vertex identifier> (*in* | *out*) *<vertex identifier>*)

  adds connecting (merging) rule in the rule.

– Directive AddConnectingRule <vertex identifier> (<vertex identifier> link <vertex identifier>) adds rule.

– Directive GetListRules shows the list of all rules (their identifiers) of the grammar.

– Directive

  Show (*<rule identifier>* | *startGraph*)

  shows either information about rule with specified identifier or start graph of the graph grammar.

– Directive RemoveConnectingRule <rule identifier> <order number of connecting rule> removes connecting rule.

– Directive SetStartGraph <graph identifier> sets the start graph.

– Directive SetName <identifier> defines name for the grammar.

– Directive Save saves grammar in the file.

– Directive Load <identifier> loads grammar from file (<identifier> sets filename without extension).

– Directive Derivate <depth of the derivation tree> builds derivation of the graph grammar with specified depth of the derivation tree to avoid circularity.

Representing grammars in software-based network models

The performance of the algorithms of self-management and program logic in the first place depend on representing basic data structures. So a program operates with graph grammars, i.e. data with complex structure, the problem of their representing became top priority task.

To implement graphs list representation was chosen. It has some advantages in comparison with matrix representations. First of all, it is rather quick way to remove and paste vertexes that are the one of the most basic actions for getting graph grammar derivation. The second advantage is memory saving, because matrixes are often sparse. After object-oriented decomposition we get three classes (*Graph*, *Node*, *Arc*) for representing such data structure as graph. By his nature graph grammars are more complex objects. It follows even from this fact that graphs are included in graph grammars. Three classes (*GraphGrammar*, *Rule*, *CRule*) can be assigned for representation these structures.

The basic class is GraphGrammar. It has some properties and methods:

```
class GraphGrammar
  {
    public string name = "";
    public Graph startGraph;
    Set <string> T, N;
    public List <Rule> rules = new List <Rule>();
    public List <Graph> derivation = new List <Graph>();
                    // methods
    }
```

Property *startGraph* contains link on start graph; *T* and *N* are alphabets of terminal and nonterminal labels respectively; field *rules* is a list of object of class *Rule*, which is described below. Graphs will be added in list derivation during the process of graph grammar derivation. It will be necessary in future for checking results of work of grammar.

Now describe class *Rule*:

```
class Rule
  {
    public string name = "";
    public Graph left;
    public Graph right;
    public List <CRule> crules = new List <CRule>();
  }
```

Left and right side graphs can be assigned as well as in mathematical representation of graph grammar. Also class *Rule* contains the list of merging rules (*connecting rules*), which are defined by class *CRule*.

```
class CRule
  {
    public string SourceId1;
    public int Dir1;
    public string TargetId1;
    public int State;
    public string SourceId2;
```

```
        public int Dir2;
        public string TargetId2;
        public string LinkSource;
        public string LinkTarget;
    }
```

Every object of this class is a merging rule and its properties can be interpreted like this:

"*If State* (1-*exist*, 0-*absent*) *edge SourceId*1 *dir*1 (1-*from*, 0-*in*) *TargetId*1 *then add edge SourceId*2 *dir*2 (1-*from*, 0-*in*) *TargetId*2".

Here *TargetId*1 and *TargetId*2 are labels of vertexes which exist on right side of the rule, but *SourceId*1 and *SourceId*2 are labels, which absent on the right side of rule.

A merging rule can be defined by another way. For this purpose the properties *LinkSource* and *LinkTarget* exist. If some labels are bound with them then program will be try to connect pasted part with the rest graph by them; *LinkSource* is a vertex label in the pattern graph, and *LinkTarget* is a vertex label in the pasted graph. Therefore, if before changing the edge is connected with the vertex with label *LinkSource* then this edge must be also in vertex with the label *LinkTarget*. This way is less flexible, but more handy.

Class *GraphGrammar* also contains some methods including methods for graph grammar derivation, saving in file and loading from file and so on.

Procedures of work with grammars for the algorithms of self-management

One of the tasks of the self-management of the autonomic network is to check belonging current network graph $G(t)$ to the set of acceptable graphs $G(t) \in \Omega_p$. Since, the set of acceptable graphs Ω_p is specified by graph grammar, the solving of this task is based on algorithm of parsing.

To implement derivation of the graph grammar, three base algorithms are required: Subgraph isomorphism, replacing a found subgraph by a substitutional graph, an algorithm of graph grammar derivation.

```
flag = false
k = 0
ISOMORPH($\varnothing$)
if flag then return ( $G_X \approx G_Y$, correspondence $i \leftrightarrow f_i$ )
    else ( $G_X \neg\approx G_Y$ )
procedure ISOMORPH(S)
    k = k+1
    if $S = V_Y$ then flag = true
    for $v \in V_Y \setminus S$ while not flag do
        if MATCH then { $f_k = v$; ISOMORPH($S \cup \{v\}$) }
    k = k+1
return
procedure MATCH
    [give out true, if vertex $v \in V_y \setminus S$ can be matched with vertex $k \in V_X$]
return
```

Replacing algorithm

Let we have links on vertexes and edges, which are needed to replace. And let we have a graph, which should substitute. Then:

1. Save information about the edge relations of the removing part of the graph to the rest graph (or copy graph) and then remove all corresponding vertexes and edges.
2. Paste graph from the right side of the rule into the source graph.
3. Apply merging rules.

Merging rules can be applied as follows

1. Find in the copied (unchanged) graph the vertex with identifier *SourceId*1. This vertex mustn't be among the vertexes which are subject to remove. If such vertexes don't exist then stop the process.
2. Then find the vertex with identifier *TargetId*1 among removed vertexes (for this target also use copied graph). If such vertex wasn't found then the process go back to step 1.
3. Depending on *dir*1 and *State* check existence or absence of the edge between these vertexes.
4. If condition wasn't satisfied then go to step 2. Otherwise go to step 5.
5. Find in the copied (unchanged) graph the vertex with identifier *SourceId*2. This vertex mustn't be among the vertexes which are subject to remove. If such vertexes don't exist then stop the process.
6. Then find the vertex with identifier *TargetId*2 among the vertexes, which were pasted. If any vertex wasn't founded then the process go back to step 5.
7. Add an edge and go back to step 6.

In the case of a merging rule is specified by labels of the vertexes which are needed to be linked between each other, the merging rules application algorithm will be next:

1. Find the vertex with the label *LinkSource* in the unchanged graph. This vertex must belong to the set of vertexes which will be removed.
2. Find the vertex with label *LinkTarget* in the changed graph, and this vertex must be among those, which were added.
3. If any edge enters in the vertex with label *LinkSource* then add this edge to the finite graph, replacing vertex with the label *LinkSource* by vertex with the label *LinkTarget*.
4. Execute the process for the outgoing edges in much the same way.

As a result, after executing of the algorithm, the graph, which will became a part of procedure of the graph grammar derivation, included in the algorithm of recognition of belonging the autonomic network graph to the set of acceptable graphs, will be created.

Solving graph connectivity problem by graph grammars

As stated above, one of the problems which can be solved by graph grammars is the problem of graph connectivity. Graph grammar, making arbitrary undirected disconnect graph connected by minimum number of edges, is presented next. The source graph being used as a start graph of this graph grammar. All vertexes in the source graph must have identifier “*v*”.

The first rule (figure 1) chooses vertex and make it connected (i.e. set it identifier “*c*”). This rule used only on the first step of the derivation. It can be removed if before the application of the graph grammar the identifier “*c*” will be set for a random vertex of the source graph.

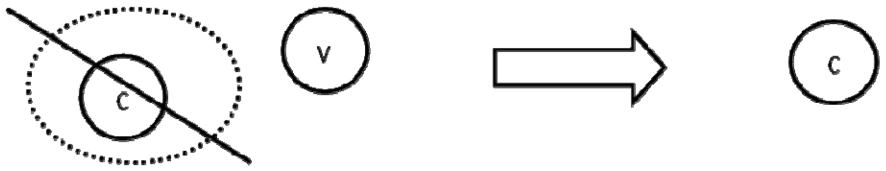

Figure 1: Starting rule of the graph grammar

The second rule (figure 2) labels adjacent vertex as connected. And when all except one vertexes of the connectivity component are labeled the third rule (figure 3) is applied. The identifier of the last vertex is set "*e*".

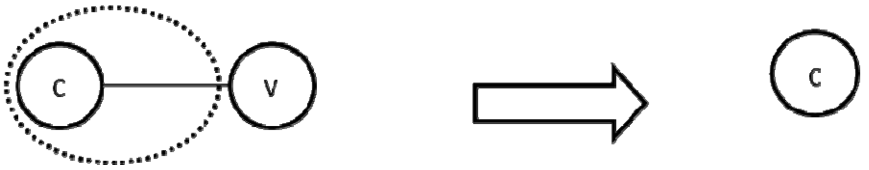

Figure 2: Rule for labeling connected vertexes

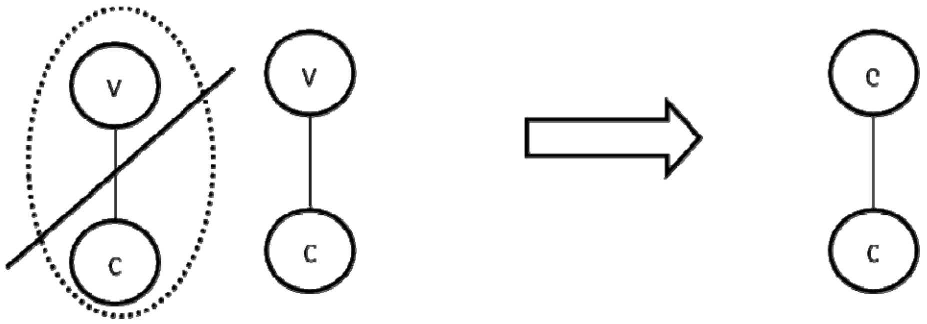

Figure 3: Rule for checking last vertex of the connectivity component

When all vertexes of the connectivity component are passed the fourth rule (figure 4) makes the edge from vertex with identifier "*e*" to any isolated vertex. And the fifth rule (figure 5) makes such edge from any vertex with identifier "*c*" (it's necessary to give all possible connected graphs). Using identifier "*t*" is caused by the necessary to avoid ambiguity in describing of merging rules. This identifier is temporary and replaced by the sixth rule (figure 6).

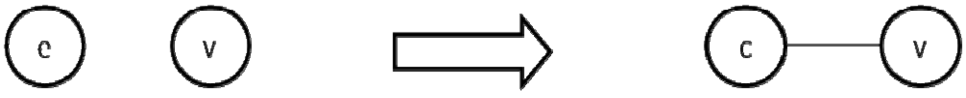

Figure 4: Adding edge to isolated vertex (from last vertex)

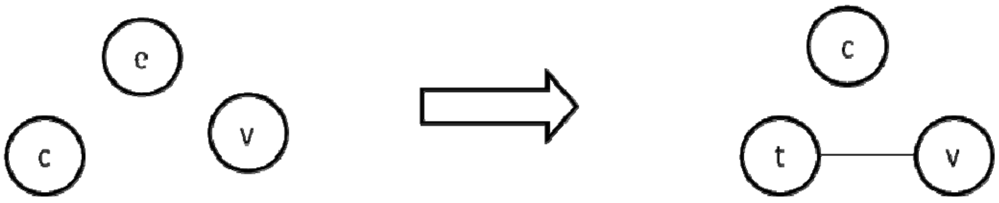

Figure 5: Adding edge to isolated vertex (from any vertex)

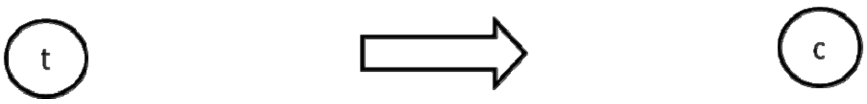

Figure 6: Renaming the vertex

This graph grammar can't give correct results in the case of empty (fully disconnected) graphs. That's why the seventh rule (figure 7) was designed. This rule will apply only if there is now any vertex with identifier "*c*", which connected with any other vertex, in the graph.

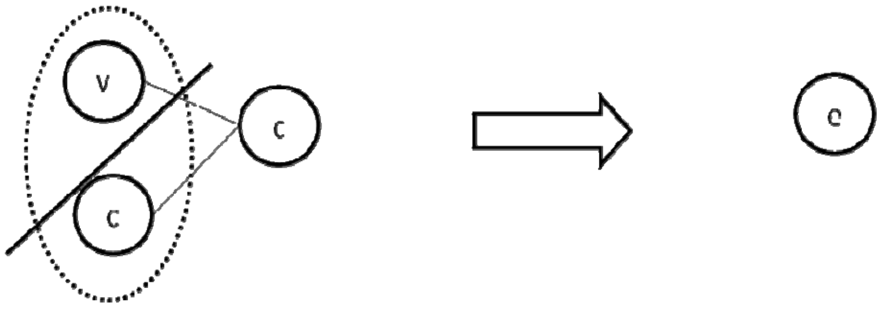

Figure 7: The rule for empty graphs

The leaves of the derivation tree of this graph grammar will be connected graphs built on basis of the source graph (graph grammar start graph) by adding minimum number of edges as it possible (connected component number minus one). The various cases of component connecting are possible, from case, when all components are connected with one (central component), to sequential connecting of the components. Disadvantage of this grammar is the large size of the derivation tree containing a priori unnecessary, redundant branches. In practical using the first and sixth rules can be removed, that improves performance of this grammar.

Adaptation of a self-managed computer structure

Providing the permanent correspondence between variable structure *G*(*t*) and solving by the system task is only a part of the self-management problem. The mentioned structure changes arise from the inner reasons: nodes breakdown, appearing of the new nodes, changes in the routes and in the connectivity because of nodes moving or battery charge decreasing/recovery [7]. The autonomic computer network must have such property as adaptation. This means that change of the outer conditions, change of the problem class, solving by the network, can lead to change of the acceptable structure class and accordingly to change of the graph grammar [8].

The architecture of such system includes two control loops: inner and outer.

The target of the inner loop functioning is permanent (periodic) control of the correspondence between system structure and graph grammar *GG*. Since grammar specifies the regularity of the network structures class to solve some general problem, the setting of the network structure for specific task and providing scalability are included in this target. Depending on the scale of the problem (it is difficult) the network structure of the different scales (amount of nodes and edges) can be generated by grammars. Thus in this case grammars can be used not only for the checking the structure, but also for generating it (generative graph grammar).

The target of the outer loop is transforming the graph grammar *GG*, setting it for the problem class. For this purpose operations under graph grammars, by which algorithm, getting the general task description (from wide class) as parameter and giving out the graph grammar of the autonomic computer system structure on the exit, are introduced.

Conclusion

Touched in the article topic is on the border of such fields of research as the theory of computational processes, the graph (and more complex structures) theory, the theory of automatic management. In the last time the theory of mobile processes are developed by R. Milner [9] and his followers [10] with using the category theory, conception of the bigraph and bisimulation.

The next advancement as regards analytical aspect is connected with the enhancement graph grammars to more complex formal systems, describing adequately not only structural, but also behavioral aspects. As for implementation grammars in the models of the autonomic systems it is required development of the parallel and distributional algorithms of the derivation.

Acknowledgement

The paper is published with support by Russian Foundation for Basic Research (project number 12-07-00763-a).

Bibliography

[1] O. Andrei, H. Kirchner. A Higher-Order Graph Calculus for Autonomic Computing. In: Graph Theory, Computational Intelligence and Thought, P. 15-26. Ed. M. Lipshteyn, V. E. Levit, R. M. Mcconnell. Springer, Heidelberg, 2009.

[2] P. Bottoni, F. De Rosa, K. Hoffman, M. Mecella. Applying algebraic approaches for modeling workflows and their transformations in mobile networks. In: Mobile Information Systems, Vol. 2, Issue 1, P. 51-76, IOS Press Amsterdam, The Netherlands, January 2006.

[3] M. Saksena, O. Wibling, B. Jonsson. Graph Grammar Modeling and Verification of Ad Hoc Routing Protocols. In: TACAS'08/ETAPS'08 Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems, P. 18-32. Ed. C. R. Ramakrishnan, J. Rehof. Springer, Heidelberg, 2008.

[4] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. Fundamentals of algebraic graph transformations, Springer, 2006

[5] L. Lambers, H. Ehrig, F. Orejas. Conflict detection for graph transformation with negative application conditions. In: ICGT'06 Proceedings of the Third international conference on Graph Transformations, P. 61-76. Ed. A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, G. Rozenberg. Springer, Heidelberg, 2006.

[6] A. Habel, R. Heckel, G. Taentzer. Graph grammars with negative application conditions. In: Fundamenta Informaticae – Special issue on graph transformations, Vol. 26, Issue 3-4, P. 287-313. Ed. A. Skowron, G. Engels, H. Ehrig, G. Rozenberg. IOS Press Amsterdam, The Netherlands, June 1996.

[7] M. C. Huebscher, J. A. McCann. A survey of Autonomic Computing — degrees, models and applications. In: ACM Computing Surveys (CSUR), Vol. 40, Issue 3, P. 1-28. ACM New York, NY, USA, August 2008.

[8] I. B. Rodriguez, K. Drira, C. Chassot, M. Jimaiel. A rule-driven approach for architectural self adaptation in collaborative activities using graph grammars. In: International Journal of Autonomic Computing, Vol. 1, Issue 3, P. 226-245. Inderscience Publishers, Geneva, Switzerland, May 2010.

[9] R. Milner. Pure bigraphs: structure and dynamics. In: Information and Computation, Vol. 204, Issue 1, P. 60-122. Academic Press, Inc. Duluth, MN, USA, January 2006.

[10]F. Bonchi, F. Gadducci, B. Koenig. Synthesising CSS bisimulation using graph rewriting. In: Information and Computation, Vol. 207, Issue 1, P. 14-40. Academic Press, Inc. Duluth, MN, USA, January 2009.

Authors' Information

Alexander Mikov *– ACM Member, professor, head of the computing technologies chair, P.O. Box: Kuban State University, 149, Stavropolskaya str., Krasnodar, 350040, Russia; e-mail: alexander_mikov@mail.ru.*

Major Fields of Scientific Research: Distributed information systems, Simulation systems and languages

Alexander Borisov *– student of the Kuban State University, P.O. Box: Kuban State University, 149, Stavropolskaya str., Krasnodar, 350040, Russia; e-mail: nillerprog@gmail.com.*

Major Fields of Scientific Research: Distributed information systems, Simulation systems and languages