AUTOMATYKA • 2008 • Tom 12 • Zeszyt 2

Leszek Kotulski*

Parallel Allocation of the Distributed Software Using Node Label Controlled Graph Grammars

1. Introduction

The notion of concurrent or distributed systems has a different meaning when it is referred by the different people. Further confusion arises when we face a large variety of multiprocessor and network architectures. In the paper, we will consider distributed computing as a paradigm that allows objects to be distributed over a heterogeneous network. An object can be specified by a set of offered services (via methods provided by an object) and a set of services requested from other objects. A communication between these objects is supported by many standards such as CORBA [20], DCOM [18], ANSA [1] and allows one to create communication channels between the components, even created by different programming teams. Having a compre-hensive infrastructure on hand, we can concentrate on flexibility evolving architectures and dynamic distri-buted object creation. UML [21] simplifies distributed system description by introducing use case model and diagrammatic tools for the visualization of specified parts of a system. Graph transformations are increasingly popular as a meta-language to specify and implement visual techniques based on the UML [22, 23]. The algebraic approach (based on the graph morphism) [4] can be used to introduce visual languages definition [3], model concurrency [2] and to specify the distributed system semantics [9].

The paper focuses on constructions a system that support an allocation of parallel object-oriented applications onto a target hardware architecture. The aim of the paper is an introduction: the formal description of such a system and the way of its dynamic transformation. The use for this purpose of graph transformation is not new but the presented formalism is the one that meets the following assumptions:

– it is enough expressive to solve a problem of allocation control,
– it is sufficiently effective (i.e. its solves membership, parsing and derivation problem with polynomial time complexity),

* Institute of Automatics, AGH University of Science and Technology, Krakow

- it offers the ability of online reaction on the external events (generated by the described system environment),
- it supports distribution of graph describing the system and its transformation in a parallel way.

The inspiration of this work was one of NLC graph of grammars (ETPL(k) [8]), that solves membership and parsing problems with O(n^2) computational complexity. In section 2, we introduce aedNLC graph grammar (equivalent to ETPL(1)) that is able to describe both software and hardware structure of the specified system. The reaction of the graph transformation system on the external request generated by the environment of the specified system is (in section 3) by a finite state automata called Derivation Control Diagram. Finally the way of distribution of the allocation graph and the way of its parallel transformation is presented (in section 4). Section 5 outlines the related works and future directions of our research .

2. An indexed edge-unambiguous graph as a tool for the distributed system description

In [15], it was shown that the attributed graph is an intuitive and enough powerful mechanism for describing the current allocation state of a distributed system. The attributed and edge-labeled directed Node Label Controlled (aedNLC) graph grammar can also maintain a set of local graphs and coordinate their parallel modification in such a way that they describe a (distributed) state of the allocated system. Before introducing a parallel derivation model, some basic assumptions of the centralized allocation system should be reviewed.

The interpretation of the system model is as follows: each component of the distributed system is represented by one of the graph's nodes[1)]. Node labels describe the types of those components ("N" – computing nodes and "M" – object instances representing both processes and synchronizing them monitors); additionally, there are introduced nodes labelled by "E" and "I" defining the offered and required services.

The relations between individual components are defined by means of directed edges connecting the appropriate graph nodes. The edge label designates a type of relation between system components. We make use of the directed graph since we assume that these relations may be asymmetric. If a more detailed definition of nodes or edges is required, an adequate attribute set may be ascribed to the labels. Attributed labels keep both a structural information (basic label) and some private information associated with the given component (attributes). Attributes inside label are represented by a partial function from a set

[1)] The term "node" used in this paper is equivalent to the "graph node", while "computing node" always refers to as a computer.

of attribute names to a set of attribute values. For attributed node labels (and appropriately for attributed edge labels), we can introduce the equivalence relation $(\mu,f)\Diamond(\nu,f)\Leftrightarrow\mu=\nu$, where μ,ν are basic labels, and f is an attributing function. To simplify the notation, an underlined basic label μ (i.e. $\underline{\mu}$) represents any attributed label belonging to $[(\mu,f)]\Diamond$ equivalence class. Function $\mathsf{lab}(\mu)$ returns basic label μ. Let Σ, Γ be accordingly a set of attributed node labels and a set of edge labels, then the graph is defined as follows:

Definition 2.1

An attributed directed node- and edge-labelled graph, **EDG** graph, over Σ and Γ is a quintuple $H = (V, D, \Sigma, \Gamma, \delta)$, where

- V – is the finite, non-empty set of graph nodes, to which unique indices are ascribed. The order in the indices set defines the order within V.
- D – is the set of edges of the form (v, μ, w) where $w, v \in V$ and $\mu \in \Gamma$.
- Σ – is the set of attributed node labels.
- Γ – is the set of attributed edge labels.
- $\delta: V\rightarrow\Gamma$ – is the node labeling function. ■

For a given graph H, its components will be denoted as V_H , D_H , δ_H respectively.

In order to achieve polynomial complexity of graph parsing a graph nodes have to be indexed in an unambiguous manner and it is necessary to introduce some limitation on **EDG**-graph structure [6, 8]. The strongest of these limitations is the rule, which accepts only edges leading from a node with a smaller index to a node with a greater one, because there is no relationship between the order of component allocation and the direction of the edges; on the contrary: edge direction depends on the labels of nodes, which are connected by them. To solve the above conflict, for each label x, we introduce an opposite label –x with the following interpretation: for any nodes v, w and an attributing function f, the edge (v,(x,f),w) is equal semantically to the edge (w,(-x,f),v).

We introduce the following notation conventions: a label with double negation is identified with original label (i.e. $-(-\underline{x})$ is equivalent to $\underline{x}$), for the attributed label μ, $-\underline{\mu}$ denotes the label $(-\mu,f)$. Graph grammars can control the correctness of the such graphs derivation (called next **IE**-graphs). Let ξ denotes a set of logic formulas using attributes and labels of nodes from the **IE**-graph G, and fineset(A) denotes set of finite subsets of A. We say that the formula $\pi\in\xi$ is satisfied in a context of the graph G if π can be evaluated as true after ascribing of attributes belonging to graph G.

Definition 2.2

Triple P = (L, R, C), where:

$L = (V_L, D_L, \Sigma, \Gamma, \delta_L)$ is an **IE**-graph called left-hand side of production, with pointed node v_L ,

$R = (V_R, D_R, \Sigma, \Gamma, \delta_R)$ is an **IE**-graph called right-hand side of production,

C: $EL\times\{in, out\}\times V_L\times \mathrm{finset}(NL\times(NL\times\xi)\times EL\times\{in,out\})$ is an embedding transformation

is said to be a **production** applied to the node v of the IE-graph G.■

For any IE-graph G and any node v we can apply production P if there exists the homomorphism $h: L \rightarrow K$, where K is a subgraph of G, such that $h(v_L) = v$ and $\forall w \in V_L$ $\delta(w)) = (\delta(h(w)))$ and $\forall u \in K\ \exists z \in L$: $h(z) = u$.

Graph K is replaced by graph R according to the embedding transformation, which defines how edges connecting graph K with the rest of graph G should be replaced by edges coming from and to graph R. A new edge is generated only if formula $\pi \in \xi$ is satisfied in a context of the graph G. All the formulas introduced in the paper use only attributes of the node v and nodes which are directly connected with it (called a direct environment of v (DENV(v))), so they can be evaluated in finite number of steps.

For production P = (L, R, $\mathbb{C}$), equivalence $\mathbb{C}((\gamma, in, v) = \{(Q,(X,\pi),\mu,in)\}$means

Notation:	Informal interpretation:	Formally:
$\mathbb{C}((\gamma, in, v))=\{$ (.........)	Every edge λ labeled by "γ" and coming into node h(v) in graph G(v) (v and h(v) have the same labeling) ought to be replaced by the edge connecting the node w of the graph of the righ–hand side of the production and labeled by "Q" with the	Every edge $(p,\lambda,h(v)) \in D_G$ such that $lab(\lambda)=\gamma \wedge p \in V_G\text{-}V_K$ is replaced by the edge (p,μ,w) such that
(Q, ...) (Q,(X, π) ...)	node p of the rest graph and labeled by "X" on condition that formula π is fulfilled (for the nodes belonging to this edge).	$w \in V_R$: $\delta(w)=Q$ ($\delta(p)=X$) and π is fulfiled
(Q, (X, π),μ,..)	New edge ought to be labeled by "μ" and is coming into node w.	
(Q, (X, π),μ, in) }		

New graph $H=(V_H,D_H,\Sigma,\Gamma,\delta_H)$ created by applying the production P=(L, R, $\mathbb{C}$), with homomorphism h in the node v of the $G=(V_G,D_G,\Sigma,\Gamma,\delta)$ graph is defined as follows:

$$V_H = (V_G - h(V_L)) \cup V_R,$$

$$D_H = (D_G - \{ (p,\mu,w) \in D_G: w \in h(V_L) \vee p \in h(V_L)\}) \cup D_R$$

$$\cup \quad \{(p,\mu,w): \exists \gamma \in \Gamma, \exists q \in h(V_L), \exists d \in \{in,out\}\ (Q,(X,\pi),\mu,in) \in \mathbb{C}(\gamma,d,q) \wedge \pi(p,G)=true \wedge \delta(p)=X \wedge w \in V_R \wedge \delta(w)=Q)\}$$

$$\cup \quad \{(w,\mu,p): \exists \gamma \in \Gamma, \exists q \in h(V_L), \exists d \in \{in,out\}\ (Q,(X,\pi),\mu,out) \in \mathbb{C}(\gamma,d,q) \wedge \pi(p,G)=true \wedge \delta(p)=X \wedge w \in V_R\text{:}\ \delta(w)=Q)\},$$

$$\delta_H = (\delta_G - \{ (h(v), \delta(h(v))): v \in h(V_L)\}) \cup \delta_R.$$

Homomorphism ought to be unambiguously defined, so in the paper we assume that if a graph of the left-hand-side of the considered production consists of a single node v_L only, then the homomorphism h is defined as unique homomorphism from the node v_L to the node v (for which production is applied).

Definition 2.3

A **context dependent graph grammar** is a quintuple $\Psi = (\Sigma, \Delta, \Gamma, P, Z)$, where:

Σ	– is the finite, nonempty set of node labels,
$\Delta \subseteq \Sigma$	– is the set of terminal node labels,
Γ	– is the finite, nonempty set of edge labels,
Z	– is the initial IE-graph,
P	– is the finite set of productions (of the form defined in def. 2.2).■

3. Derivation control

We use programmable attributed and edge-labelled directed Node Label Controlled (**aedNLC**) graph grammars for controlling correctness of the allocation graph modifications. ETPL(k) graph grammar [6, 7], being a well-known subclass of grammars defined by the Rozenberg [5, 10], solves the membership problem in a very efficient way ($O(n^2)$). Such a grammar is necessary to answer a question if an **IE**-graph can be generated by its productions, for example while the composite distributed objects (called groups [12] are created. However, it does not solve our basic problem "when and how the allocation graph is transformed into another one?". Consequently, **aedNLC** graph grammar is a combination of the context dependent graph grammar and the Derivation Control Diagram (see def. 3.1).

In practice, the allocation state will change when a user of the distributed system decides to create a new component[2] ; this decision we will call an allocation request and formally such a request will be described by the function

$$\mathsf{RQ: RQ\_name} \rightarrow (\mathsf{RQ\_art\_name} \rightarrow \mathsf{RQ\_val})$$

where:

RQ_name	– is the set of allocation request names,
RQ_atr_name	– is the set of all attribute names that appear in requests,
RQ_val	– is the set of all possible parameter values.

Let *RS* denotes the set of all possible requests, so any finite subset of *RS* will be called a **requests set**. With every user request there will be associated some graph grammar productions (see def. 2.2), that force a derivation of graph H to graph G. The order of user request service and used production will be designated by the **derivation control diagram** (see def. 3.1).

We assume that users of the distributed system will enrich the **requests set**, on the other side it will be decreased during interpretation of the **derivation control diagram**. Both these actions can be made in parallel.

By analogy, function ERQ, called an external request, will represent some action of an operating system. Let *ERS* denote the set of all possible external requests, so any finite subset of *ERS* will be called an **external request set**[3].

[2] It can be also made indirectly by a program requested dynamics allocation of some components.

[3] We will not define them in the paper because they are strongly dependent on the operating system supporting computing node.

Definition 3.1

A **derivation control diagram** is a sixtuple $S = (N, I, F, T, \Pi, \text{Wait})$, where:

N – is the set of control points,

$I \subset N$ and $F \subset N$ are the set of starting control points and the set of final control points respectively,

T is a set of transitions of the form (k,q,P,SF), where:

$k,q \in N$ are control points (transition occurs between k and q),

P is either a production or $\varnothing$ symbol when no production is associated with this transition,

SF is a semantic action described with the help of three functions:

eval: $RS \rightarrow R$, which modifies attributes of the graph R (being right-hand side graph of the production P) using as the parameters the request attributes.

ρ^-: $\mathbf{finset}(RS) \rightarrow \mathbf{finset}(RS)$, which reduces the request set,

ρ^+: $\mathbf{finset}(ERS) \rightarrow \mathbf{finset}(ERS)$, which enriches the external request set;

$\Pi = \{\Pi_k, k \in N\}$ where Π_k: $\mathbf{IE}_{\mathbf{ASNET}} \times \mathbf{finset}(RS) \rightarrow T$ is a selector that for the graph G (generated by Ψ graph grammar) and the set of requests ω chooses transition of the form (k,q,P,SF),

$\text{Wait} = \{\text{Wait}_k, k \in N\}$, where Wait_k: $\mathbf{IE}_{\mathbf{ASNET}} \times \text{finset}(RS) \rightarrow \{\text{true}, \text{false}\}$ is a synchronizing function which for the graph G (generated by Ψ graph grammar) and the set of requests ω returns true if it is possible to make the transition or false when graph transformation should be delayed until either J or ω will change, so that Wait_k returns true. ■

Starting control points ($k \in I$) are always active. When synchronizing function Wait_k inside an active control point returns true then the transition is fired. If all of these evaluations fail (i.e. return false) the next evaluation process delays until some new request appears ($\omega' = \omega \cup \{r\}$).

The semantics function SF (associated with this transition):

– enriches external request set (requesting some actions of an operating system),
– removes from w the request, that is serviced ,
– evaluates parameters of the right-hand graph of the production P.

Production P is applied to the current graph G and the new graph H is created (see def. 3.4). When the activity is moved to the next control point, that is neither a starting nor a final control point, the next thread of control is created, so it is possible to concurrently evaluate the next synchronizing function associated with the starting point. When activity is moved to a final point its thread is deleted. More intuitively, a derivation control diagram can be interpreted as a graph connecting the control points (see Fig. 1) inside of which there are evaluated sequentially both the synchronizing function and the selector choosing one of the transitions from one control point to another one (drawn as an edge). During such a transition the production P_i is applied and the semantic action SF_i is executed. A graph build of the node v, nodes of the derivation control diagram which are directly connected with v and the edges connecting these nodes, is reffered to as a direct environment of v (DENV(v)).

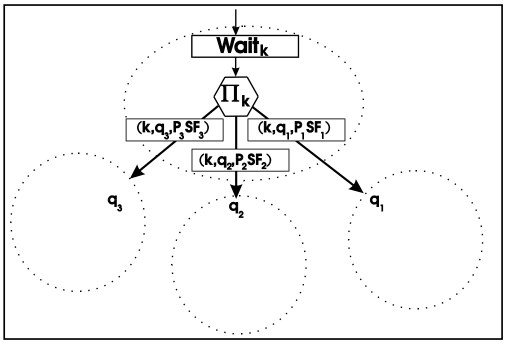


Fig. 1. Derivation graph

Definition 3.2

We say that a derivation control diagram is **effectively computable** if each selector Π_k and each synchronizing function Wait_k are evaluated:

1. using parameters at most one request u,
2. using at most attributes and graph properties direct environment of the node v in graph G, pointed by the attribute *dest_node* of the request u (i.e. (RQ(u))(*dest_node*)=v),
3. every left-hand side graph of the production is the direct environment of the node v_L.■

Definition 3.3

A pair (Ψ,S), where:

Ψ=(Σ, Δ,Γ, P, Z) is a context dependent graph grammar,

S = (N,I,F,T,Π,W) is an effectively computable derivation control diagram

is called an **attribute and edge-labelled directed Node Label Controlled graph grammar** (shortly an **aedNLC** graph grammar).■

Definition 3.4

Graph H = $(V_H,D_H,\Sigma,\Gamma,\delta_H)$ is called **directly deriverable** from a graph G by applying a production P in a node v with attributes given in a request u, when there exists a control point k, such that:

a) Wait_k(**DENV**(G,v),{u})=true,
b) Π_k(**DENV**(G,v),{u})=(k,q,P,SF),
c) semantic action SF evaluates attributes of the graph R (ie. R=eval(u)), being right-hand side graph of the production P,
d) a graph H is a result of applying the production P to the node v of the graph G. ■

We denote such a situation as H=Direct_derivation(P,v,u)(G).

Definition 3.5

Derivation control state in a context dependent graph grammar Ψ with a derivation diagram S is the quadruple C = (G, k, U, Y), where:

G – is an indexed edge-unambiguous graph (IE-graph from def. 2.3),
k – is an active control point of the derivation diagram S,
U – is a set of user requests (U∈**finset**(*RS*)),
Y – is a set of external requests (Y∈ **finset**(*ERS*)).■

A modification of a derivation control state in a context dependent graph grammar Ψ with a derivation S = (N,I,F,T,$\{\Pi_k\}_k$,$\{\mathrm{Wait}_k\}_k$) is described by the operation TRANSITION((G,k,U,Y)) defined by the algorithm presented on Figure 2 and concurrently executed operations add_RQ(U) (adds new user requests to U) and execute_ERQ(Y) (executes a request from Y and reduces it).

```
begin
while not ∃ u∈U: v=( RQ(u))(dest_node)∈V_G and Wait_k(DENV(G,v),{u})= true do end;
   U' = ρ(U);                                                ---- usually U' = U - {u};
   if  Π_k( DENV(G,v),{u})=(k,q,P,SF) ∈ T then
        Y' =ρ'(Y);
        G' = Direct_derivation(P, v, u)(G);
        TRANSITION((G',q,U',Y'));
   else ERROR("a selector evaluation fails");
    end.
```

Fig. 2. TRANSITION algorithm

The "while" loop in the operation TRANSITION is a busy form of waiting[4] until a new user request appears in U (added by add_RQ operation), for which synchronizing function Wait equals to true.

Definition 3.6

A **derivation process** in an aedNLC graph grammar (Ψ,S) is a maximal (in a sense of a length) sequence of pairs derivation control states S and transitions:

$(C_0,t_0)(C_1,t_1)......(C_i,t_i)............(C_n,t_n)$, where

for C_i = (G_i, n_i, U_i, Y_i), t_i =(n_i, q_i, P_i, (eval_i,ρ_i,ρ'_i)), G_0=Ψ.Z for n_0=0; for each *i* there exists $u_i \in U_i$, such that the atribute *dest_node* points the node v_i , which fulfils the following properties:

4 In a real environment it should be substituted by some synchronizing construction.

1) Waitn_i(**DENV**(G_i,v_i),{u_i})= true
2) Πn_i(**DENV**(G_i,v_i),{(u_i})=t_i
3) $q_i = k_{i+1}$, $\rho_i(U_i)=U_{i+1}$, $\rho'_i(Y_i)=Y_{i+1}$,
4) G_{i+1}= **Direct_derivation** (P_i, v_i, u_i)(G_i).■

The definitions introduced above represent the world of sequential computations, the formalism of an aedNLC graph grammar allows one to introduce some parallelism into a derivation of an allocation state. The basic idea is to split an allocation graph into separately maintained subgraphs and to synchronize their derivations by using requests. The modules responsible for a local derivation, will be called allocators, and represented by thenodes indexed with 0 inside each of local graphs. An example of such a solution will be presented in the next chapter.

Definition 3.7

A **parallel derivation control state** in a context dependent graph grammar Ψ with a derivation diagram S is a set of states C_1 to C_m such that:

1) C_i =(G_i, k_i, U_i, Y_i) is the derivation control state (see def. 3.5), where G_i is an IE graph.
2) there exists a communication protocol, which guarantees, that for any i, j a request generated by the allocator controlling subgraph G_i, will be served in a finite time by the allocator controlling subgraph G_j.■

Definition 3.8

A **parallel derivation process** in a aedNLC graph grammar (Ψ,S) is a maximal (in a sense of a length) sequence of the parallel derivation control states S and transitions of the form: ($\mathbf{C}_0$,t_0)($\mathbf{C}_1$,$\mathbf{t}_1$)......($\mathbf{C}_j$,$\mathbf{t}_j$)............($\mathbf{C}_n$,$\mathbf{t}_n$), where each j-th derivation control state and each j-th transition are of the form $\mathbf{C}_j=(C_{j,1},\ldots,C_{j,m})$ and $\mathbf{t}_j=(t_{j,1},\ldots,t_{j,m})$, and for each 1=p=m ($C_{0,p}$,$t_{0,p}$)...($C_{j,p}$,$t_{j,p}$)....($C_{n,p}$,$t_{n,p}$) is a derivation process (see Definition 3.6).■

4. Distributed derivation of the allocation graph

In [15] there was considered an example of a sequential derivation of the allocation graph. The minimal set of basic node labels which enables one to illustrate the connections among the distributed software application components and the hardware computing nodes may be reduced to four classes of attributed node labels:

- N – representing a computing **n**ode in a local computer network;
- M – representing an object instance which encapsulates some data structure, offers some services and requests some tasks from other object instances (a **M**onitor-like structure);
- E – representing an **e**ntry gateway to one of the services offered by an object;
- I – representing a stub module, which is a local **i**nterface to a remote service.

The relations among the system components can be defined by five classes of attributed edge labels:

- a – representing a relation "is allocated in ....", (for all edges connecting any node representing object instance with the node representing the computing node, in which this object is allocated);
- b – representing a relation "belongs to ...." (for all edges connecting any node representing entry unit with the node representing their object instance);
- c – representing a relation "potential calls ...." (for all edges connecting any node representing object instance with the node representing stub supporting the requested services);
- l – representing a relation "links with ...." (for all edges connecting any node representing stub with a node representing proper entry gateway);
- n – representing a relation "network connection" (for edges connecting nodes representing computing nodes and express possibility of hardware communication between these nodes).

For the technical reason we introduce following assumptions:

- the node labeled as A, represents the abstract allocator and is indexed by 0;
- for any edge $(0, \mu, v)$, the basic label of μ is defined as (written in small letters) concatenation of the letter "a" and the basic label of node v (i.e. am for the object instance, ai for the stubs, etc.);
- edges, which link a nodes v representing an object instances (labeled by M) with a nodes w representing a stubs offering potential services of other instances are labeled by mi or -mi.

The parallel derivation of an allocation graph, introduced in the paper, is based on the virtual nodes and edges concept. A virtual node doesn't represent any real software component described by a local allocation graph, but it keeps a necessary information to point a node representing a real software inside another local graph. The virtual node label is defined as a concatenation of the letter "V" and the appropriate basic label (on the diagram, it will be represent by the black circle). We will use two virtual nodes VE and VI.

A virtual edge is defined as an edge where one of the nodes represents a real node and the second one represents a virtual node, that points a real node in another local graph (for the simplicity we assume that edge's label will be the same as label of the corresponding virtual node).

We assume that for each virtual edge $X \rightarrow VZ$, where VZ points some node labeled by Z in the graph G_p , there exists a virtual edge $VX \rightarrow Z$ inside G_p associated with it, where node labeled by VX points the node X mentioned in the previous edge.

After introducing of productions, which allow us to construct an allocation graph we will show how the derivation control diagram mechanism helps us to synchronize a work of allocators in order to fulfill above assumption.

As the result, the indexed edge-unambiguous graph, IE-graph representing distributed system can be described with the help of basic node labels {N,M,E,I,A,VE,VI} and basic edge labels, namely {a,b,c,l,n,am,ae,ai,ve,vi,-vi,-ve,-ai,-ae,-am,-n,-l,-c,-b,-a}.

Five requests, presented below, allow us to describe a very intuitive system which supports the allocation of the distributed system components by a single allocator:

RQ(*add_node*), which demands to connect a new computing node to the distributed network;

RQ(*all_obj*), which demands to allocate an object instance pointed by *pattern* parameter with the name pointed by *name* parameter in the computing node pointed by *node_id* parameter;

RQ(*select_link*), which informs the allocator which service will be associated with the allocated object instance;

RQ(*remove_obj*), which demands an object instance to be removed from the system;

RQ(*remove_node*), which demands the removal of a computing node.

The following requests support (based on Two Commit protocol) synchronization of the distributed allocation graph by a few allocators:

RQ(*find_entry*), which asks the allocator about possibility of servicing an object request by the subsystem controlled by this allocator;

RQ(*answer_yes*), which informs that the service mentioned in the *find_entry* request can be accomplished;

RQ(*acceptance*), which confirms agreement made with the help of two previous requests;

RQ(*confirmation*), which informs about the correctness of the local allocation – parameter *commited* is set to true or false, appropriately.

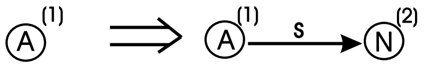
The derivation control diagram designed for both controlling local allocation process and for synchronizing creation of the distributed allocation graph consists of eight nodes (numbered from 0 to 7), where 0 is the starting control point and 7 is the final one. Table 1 defines selectors and synchronizing function for these points.

Seven productions presented below allow us to create collection of IE-graphs describing current allocation state.

For the simplicity of notation we assume that inside of all productions:

1. The symbol G is reserved for the current allocation IE -graph modified during a derivation steep and the symbol v is reserved for the node in context of which a production is applied; each graph L of the left-hand side graphs consists of a single node v_L, so the homomorphism h is defined as unique homomorphism between v_L and v.
2. Attributes of the node indexed by 0, called global attributes, are written by capital letters.
3. A global attribute CN represents the second node (different from $h(v_L)$) of the edge currently replaced by the embedding transformation (during interpretation of $C(\gamma,\mathrm{out}, v_L)$ for the edge $(h(v_L),\gamma,u)\in D_G$ attribute CN points node u).

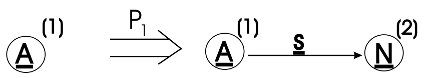
4. A value of the attribute a in node w is described by a_w.
5. A function Allocated_in:V→V, which for any node representing software component returns the node representing computing node in which it is allocated; correctness of this function is based on assumption that an object instance can be allocated only inside one computing node.
6. An operation COPY_REST, which makes the copies of all edges connecting v with the rest of graph G (and as a consequence removed from the derivered graph) and not mentioned in the embedding transformation with one modification – instead of v the node of the right-hand side production indexed by 1 is placed.
7. A production can be represented graphically



where: ⇒ separates left-hand side graph from right-hand graph, → represents directed edges, labels are inside nodes and above edges, indices of nodes are in parentheses.

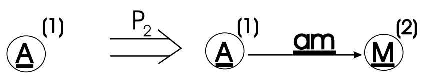
Now we will introduce seven productions, that allow to create the graph representing the current allocation state of the controlled distributed system.

The initial allocation IE-graph consists of a single node labelled by "A" (Z from def. 2.3).



Next we should be able to enrich hardware environment. The production P_1 introduces in the allocation graph G a new node (labeled by N) and associates it (edge labeled with s) with the node 0 (representing the allocator). The embedding transformation is defined as:

$\mathbb{C}_1$ = { ((s,out, v_L), { (A,(N,true),s,out), (N,(N,true),-n,in), (N,(N,true),n,in) }),
COPY_REST) }



When we would like to allocate an instance of an object in the computing node NODE we must apply the production P_2 with the embedding transformation defined as follows:

$\mathbb{C}_2$ = { ((s,out, v_L), {(A,(N,true),s,out), (M,(N,π_1), -a, in) }),
((ai,out, v_L), {(A,(I,true),ai,out), (M, (I,π_2), -mi, in) }),
((ai,out, v_L), {(A,(VI,true),ai,out), (M, (VI,π_2), -mi, in) }),
COPY_REST) }, where

π_1: NODE = *name*$_{CN}$,

π_2: (*req_servive*$_{CN}$, *req_type*$_{CN}$)$\in$ **ENV**(m)$|_{RRQ} \wedge$ **Allocated_in**(CN)=NODE)

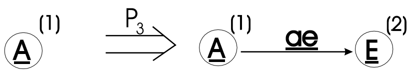
CN represents a stub to one of the requested services, this stub is allocated in computing node pointed by NODE.

Table 1

Synchronizing function		x				
point	condition	condition inside the selector				SF
	Exists any request rq	rq = *create_allocator*				Create new allocator
		rq = *ad_node*			P1	–
		rq = *all_man*			P2	–
		rq = *find_entry*				send(*answer_yes*
		rq = *acceptance* & proper node exists			Px3	send(*confirmation*(true...
		rq = *acceptance* & proper node doesn't exist				send(*confirmation* (false...
	–	For each object's entry point			P3	–
		All entry points has been visualized			–	–
	–	Exists not linked object's request and some proper stub has been already allocated			P4	–
		Exists not linked object's request and proper virtual stub has been already allocated			P4v	
		Exists not linked object's request and none stub exists				
		All object's requests have been linked				
	–	Stub will represent a local object			P5	–
		Stub will represent an object descriebed by another allocator			–	broadcast(*find_entry*(....
	Exists any request rq of the *answer_yes* type	rq can be accepted			–	send(*acceptance*(...
		rq can not be accepted and next answer is expected				
		rq can not be accepted and it is the last answer			–	
	Exists any request rq of the *confirmation* type	$commited_{rq}$ = true			Px2	–
		$commited_{rq}$ = false			–	–

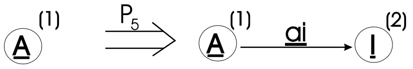
While allocating an object we must explicitly define the services which it offers. So we should remember (in the global variable AM) the index of a node representing an allocated object (generated in the previous production) and for each offered service (i.e $\forall\ e_i \in \mathrm{ENV}(m)|_{SRV}$) we will execute the production P_3. The embedding transformation is defined as:

$\mathbb{C}_3 = \{$ ((am,out, v_L), {(A,(M,true),am,out), (E, (M,π_3),-b,in) }), COPY_REST) }, where π_3: CN=AM.



For all object requests we ought to link it with some stub representing a remote service of another object. If such a service already exists, we use the production P_4, which has the following embedding transformation:

$\mathbb{C}_4 = \{$ ((-mi,in, v_L), {(M,(I,π_4), -c, in), (M,(I,not π_4), -mi, in)}), COPY_REST }, where π_4: CN=X



The production P_5 which allocates a new stub, has the following embedding transformation:

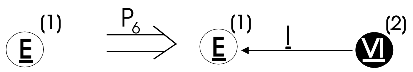
$\mathbb{C}_5 = \{$ ((ae,out, v_L), {(A,(E,true),ae,out), (I,(E,π_5), -l, in) }),
((am,out, v_L), {(A,(M,true,am,out), (I,(M,π_6),c,in) }),
COPY_REST }, where π_5: CN=X $\wedge$ π_6: CN=AM

The last production is not intuitive because the main action (adding the edge between the allocated object and node of right hand side indexed by (2)) is described by the embedding transformation (with respect of its connection with the node representing an allocator, by an edge labeled by am, it will be also connected with the node labeled by I indexed by (2)).

When the currently allocated object instance (maintained by the allocator A_S) requests a service offered by an object instance created by allocator A_T, then the production P_6 is applied to allocation graph maintained by A_S (after *acceptance* of service request) and the production P_7 is applied to allocation graph maintained by A_T (after *confirmation(true,...)* service request).

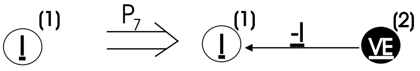
The production P_6 which allocates a new stub, has the following embedding transformation:

$\mathbb{C}_6 = \{$ ((ae,in, v_L), {(E,(A,true),ae, in), (VI, (A,true),ai, in)} }.



The production P_7, which allocates a new stub, has the follo-wing embedding transformation:

$\mathbb{C}_7 = \{ \quad ((ai,in, v_L), \{(I,(A,true),ai, in), (VE, (A,true),ae, in)\} \}$.



We assume that attributes of virtual nodes VI and EV unambiguously bind them with nodes E and I inside their local allocation graphs, so any modification of VI→E or I→VE edges can be made in cooperation of the both allocators.

5. Conclusions

A distribution (and a partial replication) of the information concerning the way of allocation of the software components into a global computer network is necessary with respect to improvement of allocation efficiency and system reliability. The example of formal description in such a situation, presented in the paper, seems to be specially interesting in context of an increasing complexity of software and hardware levels. Using graph transformation to support online control of software allocation leads us to the question of the computational complexity of the tool supporting the solution. The aedNLC graph grammar has linear computational complexity of derivation of next allocation graph, considered here as local graphs, depending on the number of nodes in the (local) graph (d_p). The introduced protocol depends linearly on the number of local graphs (k). If we denote $n=\max(k,d_1,d_2,\dots d_k)$ then the final complexity is linear with respect of n (i.e. O(n)). Let us note that a dynamically developed infrastructure of heterogeneous networks increases the influence of parameter k on a final system effectiveness. The usefulness of the aedNLC graph in the following applications has been presented: for supporting of the agents migration in [4], and for the distributed adaptive design in [13].

One of the primary requirements for the analyzed allocation construction is the capability of enforcing modularity in definition of the allocation behavior [24]. Group concept [12, 17] allows us to represent part of a distributed system as one complex object distributed over many computing nodes. The internal group structure is described by IE graph, so during its allocation we need to parse this graph. Parsing in aedNLC graph grammar has $O(d^2)$ computational complexity, so in this case the allocation has also polynomial computational complexity. We outline here the polynomial complexity of parsing, membership and derivation algorithms in the proposed graph transformation scheme, because of well known computational complexity problems in most of graph transformation systems. This is the main reason why very elegant DIST(GRAPH) notion [22, 23] based on algebraic (push-out) approach is used only to specification of the distributed system behavior. Moreover, the whole graphs structure (network and local graphs) is maintained in one place, what can create both efficiency and reliability problems during the allocation of the software in WANs.

The online reaction on the external events is not new (see GRACE (Kreowski at all 2001) but it should be noted that introduced in the paper cooperation of DENV (described as finite state automata with some synchronization rules) with events (represented by request sent as a message) enable describe these cooperation in a distributed manner.

Finally, it can note that, the presented solution can be used to solve other problems, which can be described as an intensive local graphs transformation and coordination of local graph consistency made from time to time. Such a computation model appears in pattern recognition and distributed data mining.

References

[1] APM 1991: *ANSAware Release 3.0 Reference Manual*. Architecture Projects Management Ltd, Poseidon House, Castle Park, Cambridge 1991.

[2] Baldan P., Corradini A., Ehrig H., Löwe M., Montanari U., Rossi F., *Concurrent Semantics of Algebraic Graph Transformations.* In: Ehrig, Kreowski, Montanari & Rozenberg (Eds), Handbook of Graph Transformation, vol. 3, 1999, Word Scientific.

[3] Bardohl R., Taentzer G., Minas M., Schürr A., *Application of graph transformation to visual Language.* In: Ehrig, Engels, Kreowski & Rozenberg (Eds), Handbook of Graph Transformation, vol. 2, 1999, Word Scientific.

[4] Ehrig H., Heckel R., Korff M., Löwe M., Ribeiro R., Wagner A., Corrardini A., *Algebraic Approaches to Graph Transformation II: Single Pushout and Comparison with Double Pushout Approach.* In: G. Rozenberg (Eds), Handbook of Graph Transformation, vol. 1, 1997, Word Scientific.

[5] Engelfriet J., Rozenberg G., *Graph grammars based on node rewriting: an introduction to NLC graph grammars*. LNCS, 532 (1991), 12–23.

[6] Flasiński M., *Characteristic of edNLC-graph Grammars for Syntactic Pattern Recognition.* Computer Vision, Graphics and Image Processing, vol. 42, 1989, 1–21.

[7] Flasiński M., *On the Parsing of Deterministic Graph Languages for Syntactic Pattern Recognition.* Pattern Recognition, vol. 26, 1993, 16–93.

[8] Flasiński M., *Power Properties of NCL Graph Grammars with a Polynomial Membership Problem*. Theoretical Computer Science, vol. 201, 1998, 189–231.

[9] Heckel R., Küster J., Taentzer G., *Confluence of Typed Attributed Graph Transformation Systems.* In: Corradini, Ehrih, Kreowski & Rozenberg (Eds.), 1st Int. Conference, ICGT 2002, Barcelona, Spain, Springer LNCS 2505, 2002.

[10] Janssens D., Rozenberg G., Verraedt R., *On Sequential and Parallel Node-rewriting Graph Grammars*. Computer Graphics and Image Processing, vol. 18, 1982, 279–304.

[11] Kotulski L., Flasiński M., *On the Use of Graph Grammars for the Control of a Distributed Software Allocation*. The Computer Journal, vol. 35 , 1992, A167–A175.

[12] Kotulski L., Jurek J., Moczurad W., *Object-Oriented Programming in the Large Using Group Concept.* In: Computer Systems and Software Engineering – 6th Annual European Conference, Hague 1992, 510–514.

[13] Kotulski L., Strug B., *Distributed Adaptive Design with Hierarchical Autonomous Graph Transformation Systems*. ICCS 2007, LNCS 4488, Beijing(China), 880–887.

[14] Kotulski L., *Supporting Software Agents by the Graph Transformation Systems*. V.L. Alexandrow et al (eds), ICCS 2006, LNCS 3993, Reading (UK), 887–890.

[15] Kotulski L., *Model systemu wspomagania generacji oprogramowania współbieżnego w środowisku rozproszonym za pomocą gramatyk grafowych.* Postdoctorals Lecturing Qualifications, Jagiellonian University Press, 2000, ISBN 83-233-1391-1.
[16] Kreowski H.-J., Busatto G., Kluske S., *GRACE as a unifying approach to graph-transformation--based specification.* Electronic Notes in Theoretical Computer science vol. 44, no 4, 2001.
[17] Magee J., Kramer J., Sloman M., *The Conic Support Environment for Distributed System.* In: Distributed Operating System – Theory and Practice, 1989.
[18] Microsoft, 1996. *Distributed Component Object Model Protocol—DCOM/1.0 (MSDN Library, Specifications).*
[19] NATO ASI Series F, 28.
[20] OMG 2004: *Common Object Request Broker Architecture (CORBA/IIOP)*, 02-12-2002, version 3.0.
[21] OMG 2007: *Unified Modeling Language*. v 2.1.1. www.omg.org.
[22] Taentzer G., Koch M., Fisher I., Volle V., *Attributed Graph Transformation with Application to Visual Design of Distributed Systems.* In: Ehrig, Kreowski, Montanari & Rozenberg (Eds), Handbook of Graph Transformation, vol. 3, 1999a, Word Scientific.
[23] Taentzer G., *Distributed Graphs and Graph Transformation, Applied Categorical Structures*. Special Issue on Graph Transformation, vol. 7, No. 4, 1999b.
[24] Zambonelli F., *How to achieve Modularity in Distributed Object Allocation*. ACM SIGPLAN Notice, vol. 32(6), 1997, 75–82.