ACSIJ Advances in Computer Science: an International Journal, Vol. 3, Issue 6, No.12 , November 2014
ISSN : 2322-5157
www.ACSIJ.org

ACSIJ
WWW.ACSIJ.ORG

Modeling function calls in program control flow in terms of Petri Nets

Dmitriy Kharitonov[1] and George Tarasov[1,2]

[1] Institute of Automation and Control Processes
Far-Eastern Branch of Russian Academy of Sciences
Vladivostok, 5 Radio st., 690041, Russia
demiurg@dvo.ru

[2] Far-Eastern Federal University
Vladivostok, 8 Suhanova st., 690950, Russia
george@dvo.ru

Abstract
This article presents a method for representing the C/C++ function call in terms of compositional Petri Nets. Principles of modeling function and function call in the program are described. Formal composition operations to construct program model from models of its functions and modules are also introduced. All results are illustrated on an example of real parallel program.
Keywords: *Compositional Petri Net, Control Flow, Program Model, C/C++ Programming Language.*

1. Introduction

There are many subjects in a computer science, such as static and dynamic analysis [1], program verification [2], performance analysis and debugging [3], which are based on the software program models. Construction of the model is very difficult and tedious process. It should represent all aspects of the source program that satisfy the goals of modeling, and, at the same time, it should not be too much detailed for analysis methods to be applied with useful results [4]. One should mention that model checking technique has the state space explosion problem [5]. This problem makes impossible full verification of the program, but part-by-part program verification can be done using special state space reduction methods or reducing model itself, removing unimportant details. The interested area of research is to develop methods and algorithms for automatic and semi-automatic proper model synthesis [6].

In this paper authors consider a technique to represent function calls in the control flow model of imperative programs. Representation includes actions that are performed on a both caller and callee sides. Proposed technique has two main features. At first it implies an easy possibility to develop an automatic model generator from real program source code. And in second it eliminates redundant states for reusable code parts in resulting model. Function models are aggregated in modules that can call other module functions. The whole program model is constructed as composition of the set of modules and the program entry point.

Petri nets are quite often used for modeling control flow in imperative programs [7, 8, 9, and 10]. Each transition is the action in program code that may contains a number of some operators; commands or blocks (depend on details). Each place is the state of the computational process between two actions. Often state means a current program memory values. Tokens are characterized the state of computational process execution. Their location in the places determine the execution of various actions, hence the place is often considered as a pre- and post-conditions of actions. Rules of transition firing let to visualize naturally the control flow of basic algorithmic constructions of imperative programs: condition, cycle and switch. We use concept Petri net object (or PN-object), introduced in [11] to construct program model. Each function of each module and the entire program as a whole represent an individual PN-object. Call from one PN-object to another describes control flow transfer from caller function to callee one.

Further, in this paper we give a brief background theory of PN-object calculus. After that the principles of modeling function and function call in the program are described. Then the technique of constructing a model of the program module and a complete model of program is presented. Finally, an example of a parallel program and its model in terms of PN-object is given.

2. Petri net object

Let $A = \{a_1, a_2, \dots, a_k\}$ is a set. Multiset on A is defined as a function $\mu: A \to 0,1,2, \dots$, that associates with each element of set A some non-negative integer number. Multisets are conveniently written as a formal sum

$n_1a_1 + n_2a_2 + \cdots + n_ka_k$ or $\Sigma n_i a_i$, where $n_i = \mu(a_i)$ is a number of occurrences of $a_i \in A$ in the multiset. Elements with $n_i = 0$ in formal sum are usually omitted. Union and intersection of two multisets $\mu_1 = n_1a_1 + n_2a_2 + \cdots + n_ka_k$ and $\mu_2 = m_1a_1 + m_2a_2 + \cdots + m_ka_k$ on set A are defined accordingly as $\mu_1 + \mu_2 = (n_1 + m_1)a_1 + (n_2 + m_2)a_2 + \cdots + (n_k + m_k)a_k$ and $\mu_1 - \mu_2 = (n_1 - m_1)a_1 + (n_2 - m_2)a_2 + \cdots + (n_k - m_k)a_k$, where the last operation is performed only when $n_i > m_i$ for all $1 \le i \le k$. We say $\mu_1 \le \mu_2$, if $n_i \le m_i$ for each $1 \le i \le k$, and $\mu_1 < \mu_2$, if $\mu_1 \le \mu_2$ and $\mu_1 \ne \mu_2$. If $n_i = 0$ for all i, than this multiset is denoted as $\mathbf{0}$. Also we denote $a \in \mu$ if $\exists n > 0 : (a, n) \in \mu$. Set of all finite multisets on set A is denoted as $\mathcal{M}(A)$.

Definition 1: *Petri net* is a tuple $\Sigma = \langle S, T, {}^\bullet(), ()^\bullet \rangle$, where
1. S is a finite set of places;
2. T is a finite set of transitions, with $S \cap T = \emptyset$;
3. ${}^\bullet(): T \to \mathcal{M}(S)$ is an incoming incidence function;
4. $()^\bullet : T \to \mathcal{M}(S)$ is an outcoming incidence function.

Multisets ${}^\bullet t$ and $t^\bullet$ are referred to as incoming and outcoming multisets of places for transition $t \in T$.

We will use standard graphical notation of Petri nets as a bipartite directed graph, where the places are represented by circles, and transitions – by rectangles. Places and transitions are connected by arcs representing the input and output incidence functions.

Definition 2: *Petri net object* (here and after *PN-object* or just *object* for short) is a tuple $\mathrm{E} = \langle \Sigma, \Gamma, M_0 \rangle$, where
1. $\Sigma = \langle S, T, {}^\bullet(), ()^\bullet \rangle$ is a object structure (Petri net);
2. $\Gamma = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ is a set of access points (AP), each having form $\alpha = \langle id_\alpha, \Delta_\alpha, \sigma_\alpha \rangle$, where
   - id_α is a name of access point,
   - Δ_α is a alphabet and
   - $\sigma_\alpha : T \to \mathcal{M}(\Delta_\alpha)$ is a transition labeling function;
3. $M_0 \in \mathcal{M}(S)$ is an initial marking.

Less formally PN-object is a Petri net, provided with a set of labeling functions.

To designate objects and their access points we will use capital letters E, F, G and greek letters α, β, γ . Entry $E[\alpha_1; \beta_2]$ means object E with two access points α_1 and β_2. The following graphical notation is used for objects. PN-object is drawn as rectangle, and its structure, if necessary and possible, is displayed inside a rectangle. The marking is represented as a number of tokens inside places. Object's access points are displayed as small squares on the boundary of the rectangle. The name of the access point is placed near the square (if needed alphabet is given). The label of transition is shown inside or near the transition and consists of access point name followed by colon and then label from alphabet. If it is clear from context, the access point name is not specified.

To refer corresponding components of the access point next designations will be used: $\Delta(\alpha) = \Delta_\alpha$, $\sigma(\alpha) = \sigma_\alpha$, and $id(\alpha) = id_\alpha$. The label of transition t in access point α will be defined as $\alpha(t) \equiv \sigma_\alpha(t)$.

id_α is the name that formed from simple alphabet. Examples of names: *in*, *out*, *begin*, *end* and other.

Note 1: Further in this article we will consider only those objects that have transitions labeled by the only one labeling function:
$\forall t \in T, \forall \alpha_1, \alpha_2 \in \Gamma : \alpha_1(t) \ne \mathbf{0} \wedge \alpha_2(t) \ne \mathbf{0} \Rightarrow \alpha_1 = \alpha_2$.

3. Operations on objects

Let's define a number of operations on objects that will be used later in the functions models construction.

Definition 3: Consider two objects $\mathrm{E}_1 = \langle \Sigma_1, \Gamma_1, M_{01} \rangle$ and $\mathrm{E}_2 = \langle \Sigma_2, \Gamma_2, M_{02} \rangle$, where $\Sigma_1 = \langle S_1, T_1, {}^\bullet()_1, ()_1^\bullet \rangle$ and $\Sigma_2 = \langle S_2, T_2, {}^\bullet()_2, ()_2^\bullet \rangle$. Operation of *formal union of objects* E_1 and E_2 creates new object $\mathrm{E} = \mathrm{E}_1 \oplus \mathrm{E}_2 = \langle \Sigma, \Gamma, M_0 \rangle$, where $\Sigma = \langle S, T, {}^\bullet(), ()^\bullet \rangle$ and $S = S_1 \cup S_2$, $T = T_1 \cup T_2$, $M_0 = M_{01} + M_{02}$, ${}^\bullet() = {}^\bullet()_1 \cup {}^\bullet()_2$, $()^\bullet = ()_1^\bullet \cup ()_2^\bullet$, $\Gamma = \Gamma_1 \cup \Gamma_2$.

Definition 4: Consider PN-object $\mathrm{E}_1 = \langle \Sigma_1, \Gamma_1, M_{01} \rangle$ that have two access points $\alpha = \langle id_\alpha, \Delta_\alpha, \sigma_\alpha \rangle \in \Gamma_1$ and $\beta = \langle id_\beta, \Delta_\beta, \sigma_\beta \rangle \in \Gamma_1$. Operation of *union of access points* α and β of E_1 creates new PN-object $\mathrm{E} = (\mathrm{E}_1)_{\gamma=\alpha+\beta} = \langle \Sigma, \Gamma, M_0 \rangle$ where $\Sigma = \Sigma_1$, $\Gamma = \Gamma_1 \setminus \{\alpha, \beta\} \cup \{\gamma\}$, $M_0 = M_{01}$, and $\gamma = \langle id_\gamma, \Delta_\gamma, \sigma_\gamma \rangle$, where $\Delta_\gamma = \Delta_\alpha \cup \Delta_\beta$, $\sigma_\gamma(t) = \sigma_\alpha(t) \cup \sigma_\beta(t)$. The value for id_γ can be chosen freely. Some simple algorithms are acceptable for further operations:
- $id_\gamma = \gamma$,
- if $id_\alpha = id_\beta \rightarrow id_\gamma = id_\alpha \vee id_\gamma = id_\beta$.

Operation of union of access points instead of two access points creates a new one that combines their alphabet and labeling function.

Definition 5: Consider PN-object $\mathrm{E}_1 = \langle \Sigma_1, \Gamma_1, M_{01} \rangle$, where $\Sigma_1 = \langle S_1, T_1, {}^\bullet()_1, ()_1^\bullet \rangle$ and $\alpha = \langle id_\alpha, \Delta_\alpha, \sigma_\alpha \rangle \in \Gamma_1$. Operation of *restriction of object E_1 by access point* α creates new object $\mathrm{E} = \langle \Sigma, \Gamma, M_0 \rangle = \partial_\alpha(E_1)$, where $\Sigma = \langle S, T, {}^\bullet(), ()^\bullet \rangle$ that, $S = S_1$, $T = T_1 \setminus \{t \in T \mid \sigma_\alpha(t) > \mathbf{0}\}$, $\forall t \in T \mid {}^\bullet t = {}^\bullet t_1, t^\bullet = t_1^\bullet$, $M_0 = M_{01}$, $\Gamma = \Gamma_1 \setminus \{\alpha\}$.

Restriction of object by access point deletes each transition having label from $\Delta(\alpha)$ with all adjacent arcs.

Definition 6: Consider PN-object $\mathrm{E}_1 = \langle \Sigma_1, \Gamma_1, M_{01} \rangle$ and its access points $\alpha, \beta \in \Gamma_1$, where $\alpha = \langle id_\alpha, \Delta_\alpha, \sigma_\alpha \rangle$, $\beta = \langle id_\beta, \Delta_\beta, \sigma_\beta \rangle$ and $\Sigma_1 = \langle S_1, T_1, {}^\bullet()_1, ()_1^\bullet \rangle$. Operation of *simple composition* of object E_1 by access points α and β forms new object $\mathrm{E} = \langle \Sigma, \Gamma, M_0 \rangle$, where $\Gamma = \Gamma_1$, $M_0 = M_{01}$ and $\Sigma = \langle S, T, {}^\bullet(), ()^\bullet \rangle$:
1. $S = S_1$,
2. $T = T_1 \cup T_{syn}$ where $T_{syn} = \{\mu_1 + \mu_2 \mid \mu_1, \mu_2 \in \mathcal{M}(T_1)$, and $\sigma_\alpha(\mu_1) = \sigma_\beta(\mu_2) > \mathbf{O}$, sum $\mu_1 + \mu_2$ is minimal, i.e. no sum exists $\mu_1' + \mu_2'$, that $\mu_1' + \mu_2' < \mu_1 + \mu_2$ and $\sigma_\alpha(\mu_1') = \sigma_\beta(\mu_2')\}$,
3. ${}^\bullet() = {}^\bullet()_1 \cup \{(\mu_1 + \mu_2, {}^\bullet(\mu_1)_1 + {}^\bullet(\mu_2)_1) \mid \mu_1 + \mu_2 \in T, \mu_1, \mu_2 \in \mathcal{M}(T_1)\}$,
4. $()^\bullet = ()_1^\bullet \cup \{(\mu_1 + \mu_2, (\mu_1)_1^\bullet + (\mu_2)_1^\bullet \mid \mu_1 + \mu_2 \in T, \mu_1, \mu_2 \in \mathcal{M}(T_1)\}$,
5. $\forall t \in T_{syn}\ \forall \xi \in \Gamma : \xi(t) = \mathbf{O}$.

Operation of simple composition for one object (unary form) and for two objects (binary form) is denoted accordingly

$$E = [E_1]_\beta^\alpha, \qquad E = E_1\,{}_\alpha[\,]_\beta\ E_2 \equiv [\mathrm{E}_1 \oplus \mathrm{E}_2]_\beta^\alpha$$

Operation of simple composition adds to object E_1 a number of new synchronization transitions T_{syn}. New transition are defined by multisets of symbols $\mu_1 + \mu_2$, where $\mu_1, \mu_2 \in \mathcal{M}(T)$ and have no labels. Incoming and outcoming multisets of the new transitions are calculated accordingly: ${}^\bullet(\mu_1 + \mu_2) = {}^\bullet(\mu_1) + {}^\bullet(\mu_1), (\mu_1 + \mu_2)^\bullet = (\mu_1)^\bullet + (\mu_1)^\bullet$.

Definition 7: Consider PN-object $\mathrm{E}_1 = \langle \Sigma_1, \Gamma_1, M_{01} \rangle$ and its access points $\alpha, \beta \in \Gamma_1$. Operation of *directional composition* (*unary form*) of object E_1 by access points α and β creates a new object E, that

$$E = {}_\alpha\xrightarrow{(E_1)}_\beta = \partial_\alpha([E_1]_\beta^\alpha).$$

For two objects $\mathrm{E}_1 = \langle \Sigma_1, \Gamma_1, M_{01} \rangle$ and $\mathrm{E}_2 = \langle \Sigma_2, \Gamma_2, M_{02} \rangle$ with access points $\alpha \in \Gamma_1$ and $\beta \in \Gamma_2$ accordingly, operation of *directional composition* (*binary form*) is denoted as

$$E = E_1 \underset{\beta}{\overset{\alpha}{\rightarrow}} E_2 = \partial_\alpha(E_1\,{}_\alpha[\,]_\beta\ E_2).$$

This definition implies that a unary directional composition performs two operations on an objects simple composition by AP α and β and restriction by AP α. Binary directional composition expressed in terms of unary, having done a formal union of the two original objects.

Let us formulate some properties that operations on the PN-objects have. We emphasize that properties are true in the context of the Note 1. For the cases of arbitrary transitions labeling, these properties may not hold.

Statement 1: Some properties of the operations on a PN-objects.
1. Operation of formal object union is commutative and associative:

$$\mathrm{E}_1 \oplus \mathrm{E}_2 = \mathrm{E}_2 \oplus \mathrm{E}_1,$$
$$\mathrm{E}_1 \oplus (\mathrm{E}_2 \oplus \mathrm{E}_3) = (\mathrm{E}_1 \oplus \mathrm{E}_2) \oplus \mathrm{E}_3.$$

Proof: Follows from the commutativity and associativity of union of sets and multisets operations and definition 3.
2. Union of access points operation is commutative and associative:

$$(\mathrm{E}_1)_{\gamma=\alpha+\beta} = (\mathrm{E}_1)_{\gamma=\beta+\alpha},$$
$$((\mathrm{E}_1)_{\xi=\alpha+\beta})_{\varsigma=\xi+\gamma} = ((\mathrm{E}_1)_{\xi=\beta+\gamma})_{\varsigma=\xi+\alpha}.$$

Proof: Follows from the commutativity and associativity of union of sets and multisets operations and definition 4.
3. Associativity of the restriction by access point operation:

$$\partial_\alpha\left(\partial_\beta(E)\right) = \partial_\beta\left(\partial_\alpha(E)\right).$$

Proof: Follows from definition 5. As far as subsets of labeled transitions after restrictions by α and β do not intersect, then in both cases we get the same set T, and hence the equality is fair.
4. For two objects $\mathrm{E}_1 = \langle \Sigma_1, \Gamma_1, M_{01} \rangle$ and $\Sigma_2 = \langle S_2, T_2, {}^\bullet()_2, ()_2^\bullet \rangle$ and their access points $\varsigma \in \Gamma_1$ and $\alpha, \xi \in \Gamma_2$ next equality holds:

$$E = E_1\,{}_\varsigma[\,]_\xi\ \partial_\alpha(E_2) = \partial_\alpha\left(E_1\,{}_\varsigma[\,]_\xi\ E_2\right).$$

Proof: Follows from definitions 3 and 5.
5. Associativity of directional objects composition:

$$E_1 \underset{\gamma}{\overset{\alpha}{\rightarrow}}\left(E_2 \underset{\gamma}{\overset{\beta}{\rightarrow}} E_3\right) = E_2 \underset{\gamma}{\overset{\beta}{\rightarrow}}\left(E_1 \underset{\gamma}{\overset{\alpha}{\rightarrow}} E_3\right).$$

Proof: Follows from definitions 3 and 5.

$$E_1 \underset{\gamma}{\overset{\alpha}{\rightarrow}}\left(E_2 \underset{\gamma}{\overset{\beta}{\rightarrow}} E_3\right) =$$
$$= \partial_\alpha\left(E_1\,{}_\alpha[\,]_\gamma\ \partial_\beta\left(E_2\,{}_\beta[\,]_\gamma\ E_3\right)\right) =$$
$$= \partial_\alpha\left(\partial_\beta\left(E_1\,{}_\alpha[\,]_\gamma\ \left(E_2\,{}_\beta[\,]_\gamma\ E_3\right)\right)\right) =$$
$$= \partial_\beta\left(\partial_\alpha\left(E_2\,{}_\beta[\,]_\gamma\ \left(E_1\,{}_\alpha[\,]_\gamma\ E_3\right)\right)\right) =$$
$$= \partial_\beta\left(E_2\,{}_\beta[\,]_\gamma\ \partial_\alpha\left(E_1\,{}_\alpha[\,]_\gamma\ E_3\right)\right) =$$
$$= E_2 \underset{\gamma}{\overset{\beta}{\rightarrow}}\left(E_1 \underset{\gamma}{\overset{\alpha}{\rightarrow}} E_3\right).$$

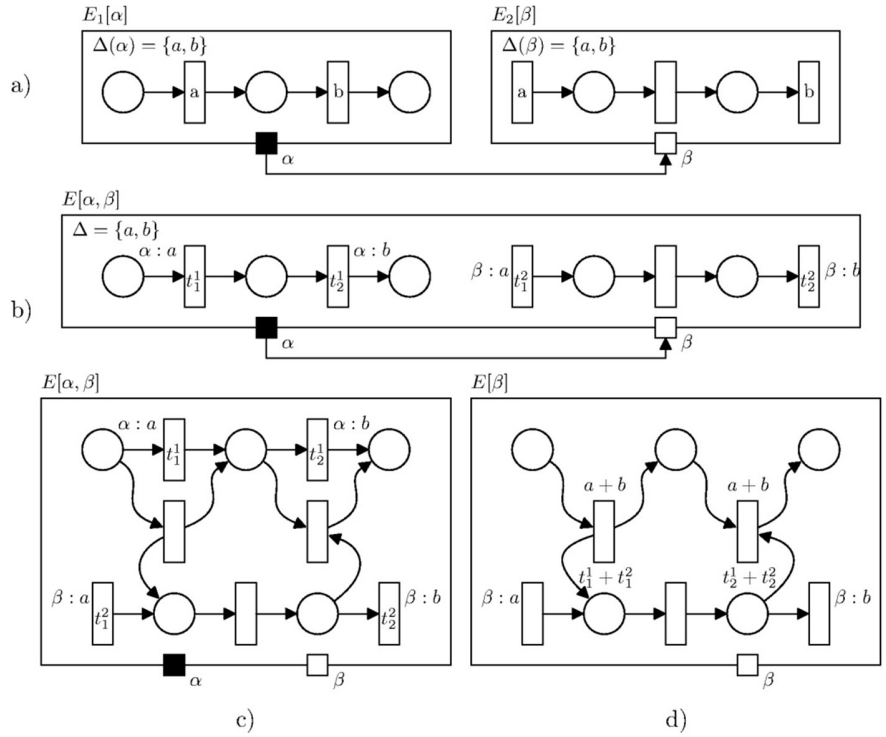

Fig. 1. Example of directional composition: (a) original representation, (b) result of objects formal union, (c) result of simple composition, (d) result of restriction.

Associativity of union of access points and directional composition of objects operations make it possible not to take into account the order of these operations on the set of original objects. As a result, to denote operation of directional composition of the object E_1, where the set of access points consists from $\Pi = \{\pi_1, \dots, \pi_n\}$ and the access point α, the next notation will be used:

$$E = \underset{\pi_n}{\overset{\left(\dots\pi_1 \overset{(E_1)}{\longrightarrow} \alpha\dots\right)}{\longrightarrow}}_{\alpha} = \underset{\Pi}{\overset{(E_1)}{\longrightarrow}}_{\alpha}.$$

We will use the following graphical notation to display PN-object operations. In a formal union of objects inside the result PN-object the original PN-objects are placed. All access points of internal objects are duplicated on the border of the external object connected with the appropriate lines. Access points for restriction operation are drawn fully filled. Operation of directional composition is represented by arrow from one access point to another. The source access point by which restriction operation is performed is fully filled. Arrow indicates the direction of synchronization: events in object E_1 can occur only if there are equivalent events in object E_2.

On the Figure 1 a sequence of operations compounded operation of directional composition of two objects E_1 and E_2 is shown. This sequence consists of formal union of source objects, simple composition by α and β access points, and restriction by α access point. The original compositional representation and result of each sub-operation are shown on a Fig. 1a)–1d) respectively.

4. Model of function and function call in imperative program

In this section we will use mathematical operations on objects described above to construct the model of the function in imperative programming languages. In such languages the concept of a *function* is often associated with traditional structured programming concept *subroutine. Function* is a certain sequence of main program actions segregated to perform the repetitive calculations. In modern programming languages the terminology associated with the concept of a *function* has become quite blurred. Different programming languages use different synonyms to identify the same entity: function, procedure, method, subroutine, subprogram, etc. Despite the number of names and according syntactic and semantic differences, the essence of the function (as a subroutine) remains the same and is a sequential execution of the next set of steps:
1. Special command function call transfers the control flow to the function while execution of commands following the call is temporarily suspended;

2. Command that forms function body run until come across special command return from function;
3. Function execution completed and control flow transfers to the next command that follows the function call.

Let's give a formal definition of the basic concepts models: function call and function.

Definition 8: Let us given a PN-object $E = \langle \Sigma, \Gamma, M_0 \rangle$, $\Sigma = \langle S, T, {}^\bullet(), ()^\bullet \rangle$ and alphabet $\Delta_f = \{begin_f, end_f\}$. Object E has access point $\alpha \in \Gamma$, in form $\alpha = \langle f, \Delta_f, \sigma_f \rangle$.

Then three elements $\langle t_1, s, t_2 \rangle$ in object E structure consisting of two transitions and places $t_1, t_2 \in T$, $s \in S$ that
1. $t_1 \neq t_2$,
2. $t_1^\bullet = s = {}^\bullet t_2, s^\bullet = t_1, {}^\bullet s = t_2$,
will be called *model of the call of function* f in the context of access point α, if

$$\sigma_f(t_1) = begin_f, \sigma_f(t_2) = end_f.$$

Less formally model of function call will denote three marked and connected in certain way net elements, where:

- first transition models control flow transfer to the function (label $begin_f$), this transition is called *call transition*;
- place models a state of function completion waiting;
- second transition models control flow return from the function (label end_f), this transition is called *return transition*.

This definition naturally describes the calls of the same function several times and the calls of several different functions in a single PN-object. In the first case, each call in the structure of the object E will have its own three elements with identical labels. In the second case, each unique function f_i call is described by a separate alphabet Δ_i and a separate access point $\alpha_i \in \Gamma$. There is general case possible, where the object E has one access point describing the calls of all functions. Using operation of union of the access points we can join all access points to one access point named, for example, *calls*. This case is shown on a Fig. 2.

Let's define the function model, assuming that the internal structure of the function control flow is already described by some Petri net.

Definition 9: Let us given PN-object $E_f = \langle \Sigma_f, \Gamma_f, M_{0f} \rangle$, where $\Sigma_f = \langle S_f, T_f, {}^\bullet()_f, ()_f^\bullet \rangle$ – object structure, describing control flow of the function body, and there is in Γ_f two disjoint subsets of access points

$$(IN_f \cup OUT_f) \subset \Gamma_f, \; IN_f \cap OUT_f = \emptyset.$$

$IN_f = \{in\}$ is a subset of incoming access points, consisting of one access point $in = \langle f, \Delta_f, \sigma_f \rangle$, $\Delta_f = \{begin_f, end_f\}$. $OUT_f = \{out_1, \dots, out_n\}$ is a subset of outcoming access points, satisfying the definition 8. If for the object structure Σ_f and for access points $in, out_1, \dots, out_n$ next statements are fair:
1. $\exists! t_b \in T : {}^\bullet t_b = \mathbf{O} \wedge \mathrm{in}(t_b) = \mathrm{begin}_f$;
2. $\forall t \in T_e : t \neq t_b \wedge t^\bullet = \mathbf{O} \wedge in(t) = end_f$, where $T_e \subset T$ and $T_e \neq \emptyset$;
3. $\forall out_i \in OUT_f : out_i(t_k) = begin_i \Rightarrow {}^\bullet({}^\bullet t_k) = t_l \Rightarrow out_i(t_l) = end_i$;
then this PN-object E_f well be referred as *function f model*.

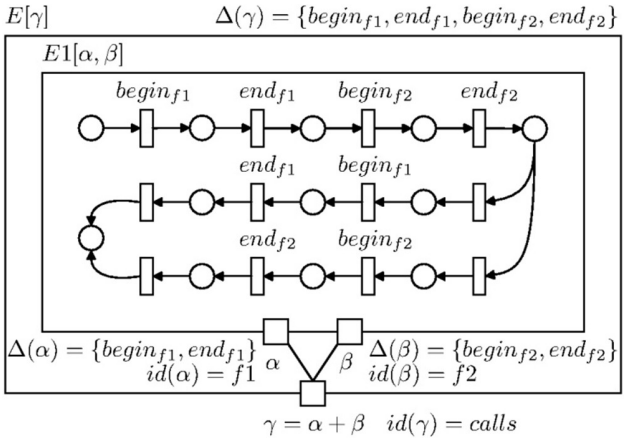

Fig. 2. Examples of function call models.

Statements 1–3 in definition 9 mean that:
1. There is only one transition t_b, that have no incoming arcs, this transition is called *incoming transition*;
2. Object structure have non-empty set of transitions T_e, having no output arcs, this transitions are called *outcoming transitions*;
3. All outcoming transitions are describing calls of functions another than f.

Let's use additional notations: $In(E_f) = IN_f$ - function In returns a set of incoming access points of PNobject, $Out(E_f) = OUT_f$ - function Out returns, accordingly, a set of outcoming access points of PN-object.

Examples of function models and function call models are shown on a Fig. 3. Objects E_f, E_{f1} and E_{f2} represent models of three different functions f, $f1$ and $f2$ accordingly. Each object has one incoming access point in_i, from which control flow is transfered to function body, modeled by object. Object E_f has two outcoming access points describing calls of functions $f1$ and $f2$. Link between caller and called functions is defined through directional composition operation by proper access points.

Process of control flow transfer can be described in a next way. Consider object E_f in the state $M = \{0,1,0,0,0,0,0\}$ (token in place s_2). Then, in object E_f there can be fired transition t_2, labeled $begin_{f1}$. In accordance with the rules of directional composition operation, objects E_f and E_{f1} are combined together, and from transitions t_3 (E_f) and t_{b2} (E_{f1}) a new transition is formed, that removes token from s_2 and puts one token to the places s_2 (E_f) and s_8 (E_{f1}). Token in place s_2 means waiting of control flow return from function E_{f1} (*awaiting token*), and token in place s_8 initiates execution of function body. Awaiting token destructs, when return transition t_4 fires, that likewise connected to outcoming transition t_{e4} in object E_{f1}.

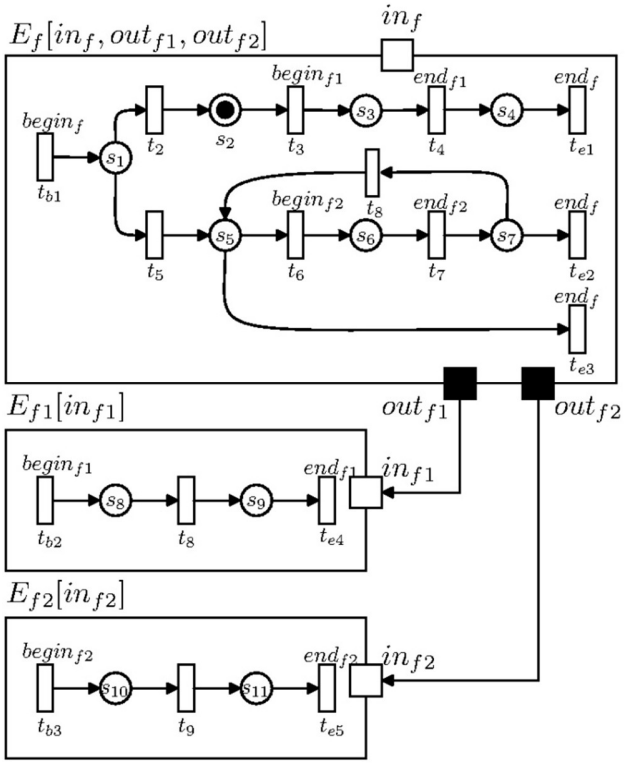

Fig. 3. Examples of function models and function call models.

Having description of functions models and function calls models let's go to program model. We will use notion *program module* or just *module* to designate some set of functions, defined in program. Let's generalize the definition of a function model to the module. The term module is consistent with such structural concepts of programming languages like library and function, and with concepts of class and method in the OOP.

Definition 10: *Module* is a PN-object $\mathrm{E} = \langle \Sigma, \Gamma, M_0 \rangle$, that have defined functions In and Out. Without loss of generality, it is assumed that

$$In(E) \cup Out(E) = \Gamma, |In(E)| = 1, |Out(E)| \geq 0.$$

Minimal module is considered to consist of only one function.

Less formally module is an object, that have only one *incoming access point*, via which can be invoked components of the module, and may have zero or more *outcoming access points*, via which it connects to other modules. Module, having empty set of outcoming access points is referred to as *full module.*

Let's introduce the operation of modules composition as a method for designing complex modules from a set of more simple ones. Let's denote for a module F subset $SELF_F \subset \Gamma_F$ that includes all output access points of the module, having alphabets included in the alphabet of input access point for this module:

$$SELF_F = \{out_i \mid out_i \in Out(F) \wedge \Delta(out_i) \subseteq \Delta(In(F))\}.$$

Access point belonging to the set $SELF$ will be called the *internal access points.*

Definition 11: Let's E_1 and E_2 are modules. Then *module composition operation* builds from two modules E_1 and E_2 the new module $\mathrm{E} = \langle \Sigma, \Gamma, M_0 \rangle$ such, that:

$$\mathrm{E} = \mathrm{E}_1 \uplus \mathrm{E}_2 = \underset{\mathrm{SELF}}{\xrightarrow{((E_1 \oplus E_2)_{in=in_1+in_2})}}_{in},$$

where

$In(E) = \{in\}$ incoming access point of new module,

$Out(E) = (Out(E_1) \cup Out(E_2)) \setminus SELF$ - set of new module outcoming access points,

in_1 and in_2 - incoming access points of PN-objects E_1 and E_2 accordingly.

Let's consider an example of two modules composition (fig. 4) M_1 and M_2, that has input access point in_1 and in_2 and subsets of outcoming access points $\{f_1, f_2\}$ and $\{f_1, f_3, g\}$ accordingly. It is assumed that via access point g a function in the second module is called from the first module. Figure 4(a) shows source modules and their access points. Figure 4(b) shows composition of source modules, specifically all the transformations of the source objects: objects are formally united in one object M, and union of access points in_1 and in_2 is performed, subset of internal access points $SELF$ is highlighted, consisting of $\{g\}$, than directional composition by access points from sets $SELF = \{g\}$ and $In = \{in\}$ is executed. As the result (fig. 4(c)) a new module M is obtained, that has all access point of source objects except for g.

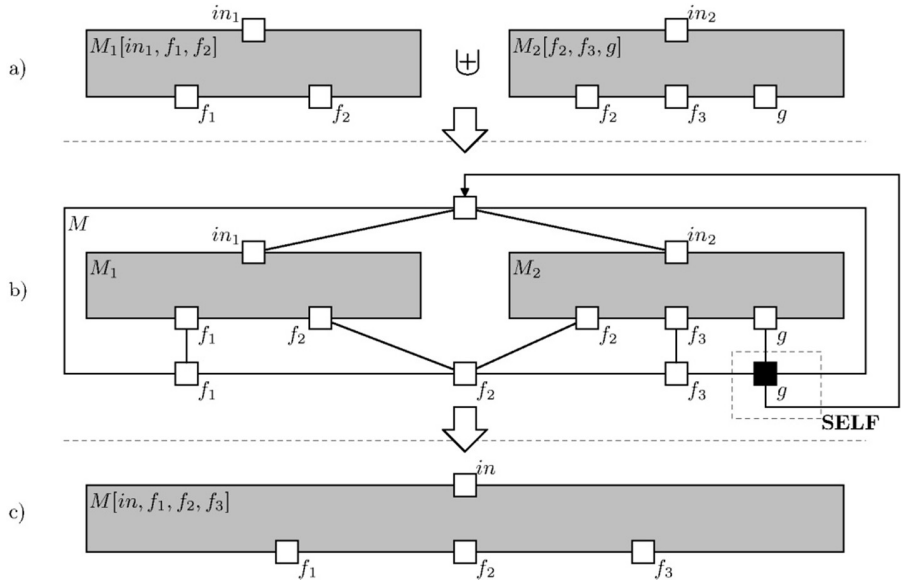

Fig. 4. Example of module composition.

Thus, applying the operation of modules composition to a certain set of program functions models one module can be obtained that contains a single incoming access point to call any function from the set and a set of output access points to call external functions. The presence or absence of the outcoming access points in the module obtained depends upon our considerations of nested function calls. For example figure 4 shows source modules having calls to some functions (f_1, f_2 and f_3). And there can be two analysis cases. In one case functions f_1, f_2 and f_3 are considered in the same way as all others and we need to include their models into the whole program model. But another way we can consider these functions as elementary, not worth scrutiny from the point of program control flow modeling. In that case each of these functions can be modeled by one transition without awaiting places, access points and so on. And thus the result of composition will be full module having only one incoming AP. The level of details is chosen by man and depends on modeling purposes.

Let's consider final program model, which execution in imperative languages begins from some *start function* (program entrance point). Before the control flow is given to this function, there is a special imperative block completed from executable file - *loader*, that compiler adds to the executable image automatically. The role of the loader is to initialize the system environment of the program and then calling the starting function.

Definition 12: Let us given PN-object $E = \langle \Sigma, \Gamma, M_0 \rangle$, where $\Sigma = \langle S, T, {}^\bullet(), ()^\bullet \rangle$ and $\Gamma = \{\alpha\}$. $\alpha = \langle out, \Delta, \sigma \rangle$ and $\Delta = \{bein, end\}$. We will call E the *model of loader*, if:

- $\exists! s_b, s_e \in S : s_b \neq s_e, {}^\bullet s_b = \mathbf{0}, s_e^\bullet = \mathbf{0}$,
- $\forall s \in S : M_0(s) = \begin{cases} 1, if\ s = s_b, \\ 0, otherwise, \end{cases}$
- $\exists! \langle t_1, s, t_2 \rangle, t_1, t_2, \in T, s \in S : s \neq s_b \neq s_e \wedge t_1^\bullet = s = {}^\bullet t_2$ and $out(t_1) = begin$ and $out(t_2) = end$.

Less formally the loader can be defined as some PN-object with highlighted initial s_b and final s_e places, modeling, respectively, the beginning and the end of the program, with a single start function call via the access point out and initial marking, having token only in the initial place s_b.

Definition 13: Let us given loader E_1 and full module E_2 with access points out and in such that:

- $Out(E_1) = \{\alpha\}$, $In(E) = \{\beta\}$, $\alpha = \langle out, \Delta_\alpha, \sigma_\alpha \rangle$, $\beta = \langle in, \Delta_\beta, \sigma_\beta \rangle$,
- $\Delta_\alpha = \Delta_\beta = \{begin, end\}$.

Then object $E = \partial_\beta(E_1 \underset{\beta}{\overset{\alpha}{\rightarrow}} E_2)$ will be called *model of imperative program*.

Example of imperative program model in terms of PN-objects is shown on a Fig. 5(b). Directional object composition operation joins these objects using access points. Restriction operation removes remaining access point in from model of module from it useless. The result is an object with no access points, the structure of which models the control flow of the program.

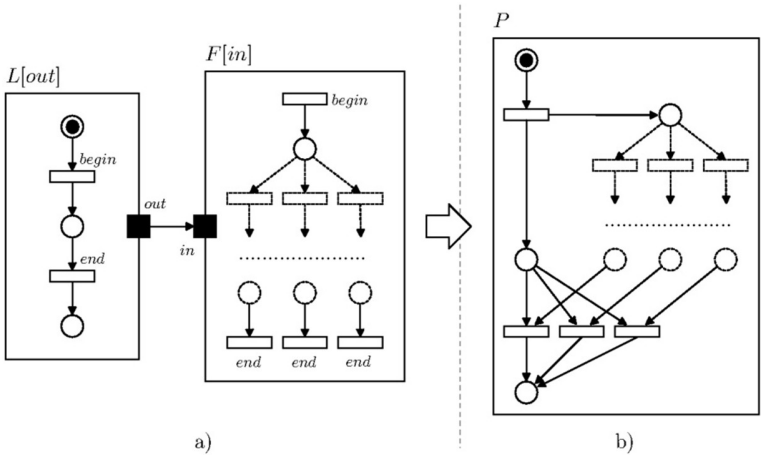

Fig. 5. A program model as an example of loader and full module composition.

5. An example of imperative C++ program model

Let's use depicted above mathematical apparatus for constructing a model of the program, written in an imperative programming language. As an example, we consider the C language, as one of the most popular general purpose programming languages, designed for a wide range of applications - from performing simple data processing and up to creation of operating systems. Briefly it can be characterized as an imperative, structured programming language. One of its distinctive features is the presence of the only start up function `main`, which is the entry point of the program, which begins execution of program instructions. In addition to the function `main`, a C program can contain any number (within an acceptable by hardware and operating system range) of other functions, with mutually different names. Nesting function descriptions are not allowed, i.e., all functions are equivalent and they can be accessed from anywhere in the program.

Let's briefly describe the scope of concepts associated with the function in C. Description of each function consists of the function header and its body. The function header describes such features as the function name, return type, and a list of the input (and output) parameters. Description of the function body follows next to the description of the function header. Body of the function is a block of statements that consist of next basic algorithmic constructions: expressions, conditions, cycles, switches, the function call operators, return from function operators, and others. Execution of statements is performed sequentially until the last statement executed, or the return from function operator meet. The transfer of the control flow to the body of the function is carried out by the construction *function call*, which is partially supported on a hardware level of CPU.

The figure 6 shows an example of the "set dividing program" and a model of its control flow. Model consists of five PN-objects. Object E_{loader} is the program loader. Initial state of the program is shown by two tokens. Each token is associated with a process, which by program design either collects larger elements of two sets ore smaller ones. Loader executes just one command - call the `main` function via transitions, labeled $begin_{main}$ and end_{main} accordingly. E_{main} object simulates common actions of each process. Transition labeled $INIT$ matches strings 33–35 of the program. After initial initialization a choice construction is performed that correspond to the rows 36–37 of the program. That construction models all possible cases of further processes execution, more precisely, either control flow transfer to object E_{small} (function `Small` call), or control flow transfer to object E_{large} (function `Large` call), or execution of bogus action $GOTO$, which is possible, if none of the conditions are satisfied.

In terms of the used Petri nets notation any of the tokens may initiate any of the transitions labeled $begin_{small}$, $begin_{large}$ or $GOTO$. However, we assume that this choice can be determinated, and one piece will excite a transition $begin_{small}$, while another - $begin_{large}$. Accordingly, function calls constructions will give control to the objects E_{small} and E_{large}. These objects have similar structures of the control flows and perform the same type of actions that differ by labels. Lines 16–23 of the program match the object E_{small} structure, lines 24–31 - the object E_{large} structure. Both objects have several calls to common interaction function `SendRecv`, the control flow of which is modeled by object E_{SR}. Rules for firing the transition $SENDRECV$ in this example are not regulated. It is

```
1  #include <mpi.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "set.h"
5
6  int rank;
7  Set SET;
8  #define SMALL_PROC 0
9  #define LARGE_PROC 1
10
11 void SendRecv(int dst, int in, int &out) {
12    MPI_Status status;
13    MPI_Sendrecv(&in, 1, MPI_INT, dst, 100,
14                 &out, 1, MPI_INT, dst, 100,
15                 MPI_COMM_WORLD, &status); }
16 void Small() {
17    int mx, x;
18    mx = SET.max();
19    SendRecv(LARGE_PROC, mx, x);
20    while (mx > x) {
21       SET - mx; SET + x; mx = SET.max();
22       SendRecv(LARGE_PROC, mx, x); }
23    SET.Write("small.txt"); }
24 void Large() {
25    int mn, y;
26    mn = SET.min();
27    SendRecv(SMALL_PROC, mn, y);
28    while (mn < y) {
29       SET - mn; SET + y; mn = SET.min();
30       SendRecv(SMALL_PROC, mn, y); }
31    SET.Write("large.txt"); }
32 int main(int argc, char *argv[]) {
33    MPI_Init(&argc, &argv);
34    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
35    SET.Init(argc, argv);
36    if (rank == SMALL_PROC) Small();
37    else if (rank == LARGE_PROC) Large();
38    MPI_Finalize();
39    return 0; }
```

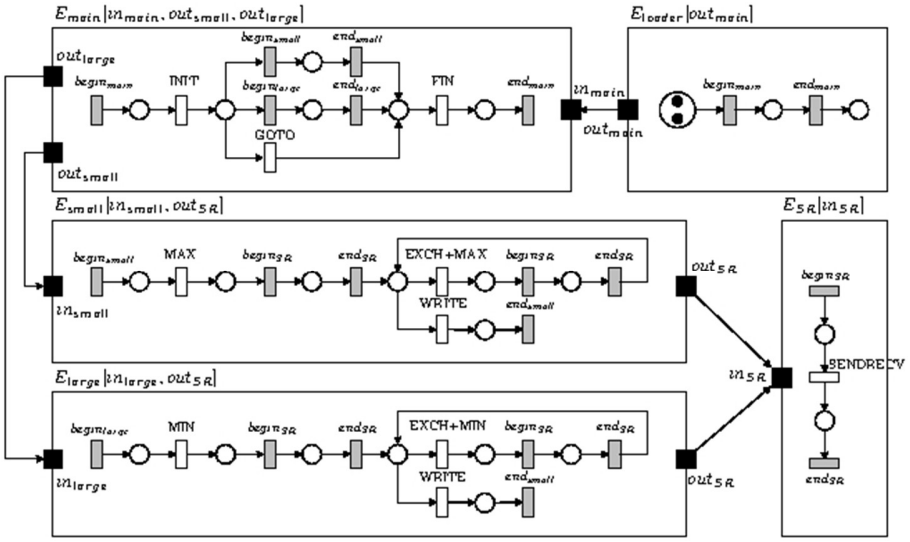

Fig. 6. Example of program and its model in terms of Petri Nets.

assumed that both tokens (both processes) excite it in accordance with the general rules of transitions firing. Synchronization of the processes can be described using function call construction and the model of the function `MPI_SendRecv` as one more PN-object.

6. Conclusions

This paper presents a method for constructing a model of imperative program control flow as the composition of control flow models of its component functions.

Using the notation of PN-object *function call* and *function body* is described. Formal directional composition of objects operation defined, that allows to obtain single PN-object modeling control flow of program from a number of smaller PN-object. Representation of the control flow in the form of function models composition have a number of significant advantages with respect to the flatten representation of one model.

In particular it partially simplifies the states explosion problem, because the number of states in the model is reducing a lot due to the fact that there are no duplicated states of the same function called from different places of the program. Stack of function calls in each moment of time is reflected by awaiting places in *function call* constructions. Then the procedure of automatic model generation from source code is simplified, because each function in program maps to its own PN-object, connected to others by a set of links. And the last advantage is that modularity of model gives more flexibility in model analysis. Input of analysis algorithms may have full compositional representation, flatten representation of program or a model of subset of program function realizing logical part or functionality of the whole program

The results obtained give us an opportunity for further successful development of the submitted approach. In particular, the work does not address how to describe recursive functions, although quick examination of the PN-objects definition and operations on them are quite suitable for compositional representation of recursion in imperative programs. Virtual functions in object-oriented programming languages are also not considered, and relate to runtime type recognition. Also there is a need to study individually the questions of description data and methods of data transfers back and forth to function.

Acknowledgments

This work was supported by the Russian Academy of Sciences (program Num. 43 of the Presidium of the RAS "Fundamental problems of the mathematical modeling"), by the Far Eastern Branch of the Russian Academy of Sciences, and by the Far Eastern Federal University.

References

[1] Ma, J., Han, W., Ding, Z.: Behavior Analysis of Software Systems Based on Petri Net Slicing. In: Huang, D.-Sh., Jiang Ch., Bevilacqua, V., Figueroa, J.C. (eds.) ICIC 2012. LNCS, vol. 7389, pp. 475-482. Springer, Heidelberg (2012).

[2] Bergenthum R.: Faster verification of partially ordered runs in petri nets using compact tokenflows. In: Colom, J.-M., Desel, J. (eds.) PETRI NETS 2013. Proceedings of the 34th international conference on Application and Theory of Petri

Nets and Concurrency. LNCS, vol. 7927, pp. 330-348. Springer, Heidelberg (2013).
[3] Kounev, S., Buchmann, A. On the Use of Queueing Petri Nets for Modeling and Performance Analysis of Distributed Systems. Petri Net, Theory and Applications, Book edited by: Kordic, V., ISBN: 978-3-902613-12-7, DOI: 10.5772/5317.
[4] Westergaard, M., Slaats, T.: Mixing paradigms for more comprehensible models. In: Daniel, F., Wang, J., Weber B. (eds.) 11th International Conference, BPM 2013. LNCS, vol. 8094, pp. 283-290. Springer, Heidelberg (2013).
[5] Camilli, M.: Petri nets state space analysis in the cloud. In: Proceedings of the 34th International Conference on Software Engineering (ICSE '12). IEEE Press, Piscataway, NJ, USA, pp. 1638-1640.
[6] Dedova A., Petrucci L.: From Code to Coloured Petri Nets: Modelling Guidelines. In: Koutny, M., van der Aalst, W. M. P., Yakovlev A. (eds.) Transactions on Petri Nets and Other Models of Concurrency VIII. LNCS, vol. 8100, pp. 71-88. Springer, Heidelberg (2013).
[7] Peterson, J. L.: Petri Net Theory and the Modeling of Systems. Prentice Hall PTR, Upper Saddle River, NJ, USA. 1981.
[8] Cherkasova, L. A., Kotov, V.E.: Structured nets. In: Gruska J., Chytil, M. (eds.) Mathematical Foundations of Computer Science. LNCS, vol. 118, pp. 242-251. Springer, Heidelberg (1981).
[9] Denaro, G., Pezze, M.: Petri nets and software engineering. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 439–466. Springer, Heidelberg (2004).
[10] Liao, H., Wang Y., Stanley, J., Lafortune, S., Reveliotis, S., Kelly, T., Mahlke, S.: Eliminating Concurrency Bugs in Multithreaded Software: A New Approach Based on Discrete-Event Control. IEEE Transactions on Control Systems Technology, vol. 21, no. 6, pp. 2067-2082, Nov. 2013.
[11] Anisimov, N.A., Golenkov, E.A., Kharitonov, D.I. Compositional Petri net approach to the development of concurrent and distributed systems. Programmirovanie, Volume 27, Issue 6, 2001, Pages 30-44.

Dmitriy Kharitonov is a senior research fellow at the Institute of Automation and Control Processes of the Far-Eastern Branch of Russian Academy of Sciences. He holds his PhD in Software of Computational Networks and Systems (Candidate of Technical Sciences, Vladivostok, Russia, 2001). He holds his MSc in Applied Mathematics (Moscow Institute of Physics and Technologies). He is interested in program verification, program modeling, parallel programming languages and translators.

George Tarasov is a research fellow at the Institute of Automation and Control Processes of the Far-Eastern Branch of Russian Academy of Sciences and Far-Eastern Federal University. He holds his MSc in Automation Control Systems (Far-Eastern Federal University, Vladivostok, Russia, 1999). He is interested in parallel programming, program verification and performance analysis.