Computing and Informatics, Vol. 36, 2017, 1063–1087, doi: 10.4149/cai_2017_5_1063

MODELING OF OBJECT-ORIENTED PROGRAMS WITH PETRI NET STRUCTURED OBJECTS

Dmitriy Kharitonov, George Tarasov, Evgeniy Golenkov

Institute of Automation and Control Processes
Russian Academy of Sciences
5 Radio st., Vladivostok, Russia
e-mail: {demiurg, george, golenkov}@dvo.ru

Abstract. The article presents a method for constructing a model of an object-oriented program in terms of multilabeled Petri nets. Only encapsulation – one of the three concepts of object-oriented paradigm – is considered. To model a different aspects of encapsulation a Petri net structured object is proposed. It consists of a Petri net defining its behavior and a set of organized access points specifying its structural properties. Formal composition operations to construct a program model from the models of its methods, classes, objects, functions, and modules are introduced and a source code translation algorithm to Petri net representation is proposed. A special section of the article considers in detail a process of model construction of a real object-oriented program (OOP). Source code of the program, figures with Petri net objects modeling different elements of the program and the resulting model of the program are presented.

Keywords: Place/transition nets, multilabeled Petri nets, program model, object-oriented programming

Mathematics Subject Classification 2010: 68N30

1 INTRODUCTION

Petri net theory is widely used in modeling and analysis of programs. The most prominent works in this scope deal with investigation of parallel program properties. Deadlock search [1, 2], program performance analysis [3, 4], verification of message

passing [5] with the help of Petri nets are just some examples of Petri nets application. We emphasize that in those and many others works, program models are of great importance providing the formal basis for the analysis of the whole program or some of its components characteristics. Alternatively, generation of an appropriate model of program investigated is the most labour intensive step of all in the program modeling process, and there was some of research about automatic creation of program models [6, 7, 8]. Keeping in mind continuous development of programming languages and environments, extensions of syntax and semantics of modern languages, the task of automatic program model construction from its source text becomes more and more significant in practice for program property analysis and verification.

This paper describes an approach and an algorithm of automatic model construction of an object-oriented program. To achieve high level of automation, an auxiliary formal construction *names tree* is introduced. From the *names tree* a PNS-object (Petri net structured object) is proposed that is assumed to be a basic unit for the complex program model construction. On the example of a "set division" program [9, 10] a step-by-step process generating a final model in terms of compositional multilabeled P/T nets is illustrated. This article deals with only one of three conceptions of object-oriented programming – encapsulation. Definitions of class declaration model, method and member declaration models and methods and functions implementation models are given. Control flow transfer by methods and function calls is considered. The proposed approach is a continuation and development of an approach which addresses issues of building models of procedural programs [11, 12].

2 C++ PROGRAM SAMPLE

To demonstrate the results of constructing a model the following real object-oriented parallel program example will be used. It is a program that solves the "set division" problem, proposed by Dijkstra in 1977 [9]. The problem was discussed in many publications of different authors, and its partial correctness was proven in [13]. In 1996 Karpov has shown the absence of the property of total correctness [10]. The problem has the following description. There are two processes, *Small* and *Large*, given sets of integers for each process, S for *Small*, L for *Large*, respectively. Process *Small* finds in its set S a maximal element and sends it to process *Large*. Simultaneously, process *Large* finds in its set L a minimal element and sends it to process *Small*. Such interaction between two processes continues until S will consist of all minimal elements, and L – all maximal elements. Cardinality of sets remains constant. The model of the algorithm of processes interactions may be schematically shown as a simple Petri net in which transitions represent transfer of minimal and maximal elements (Figure 1) by the two independent channels a and b.

The problem implementation using C++ object-oriented programming language is proposed in the Appendix. The program has two entities: the "set" and the

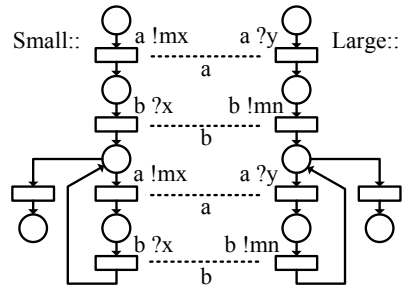

Figure 1. Schematic Petri net model of the "set division" problem

"process". The "set" behavior is realized in class `Set` and behavior of "process" in class `DivProc`. We use the MPI library for message passing between processes. It is fixed that *Small* process has rank 0, and *Large* has rank 1.

The main goal of this article is to introduce Petri net structured object notation and a set of operations, necessary for modeling of objects and their control flow in the program. As stated earlier, we consider method calling as the only method of object interaction.

When creating models of object-oriented software two technical problems should be solved. First, OO programs have a more complicated name binding mechanism than procedural programs. Any name in OOP code can be the name of a local object, class or function, or that name can be a member or method of the class that part of code belongs to, or it can be a global object or function. Thus, to build a program model automatically we need to handle all this name locations. And second, procedural programs have only function recursions while in OOP there can also be recursion through data types; classes can have methods that have local variables of the same class. Recursions prevent us from modeling classes and methods by always using copies of their models when they are used.

3 BASE DEFINITIONS

3.1 Set, Multiset, and Sequence

Let $A = \{a_1, a_2, \ldots, a_k\}$ be a *set*. A *Multiset* on set A is defined as function $\mu : A \to \mathbb{N}_0$, that associates with each element of the set A some non-negative integer number. Multisets are conveniently written as a formal sum $n_1a_1 + n_2a_2 + \ldots + n_ka_k$ or $\Sigma n_i a_i$, where $n_i = \mu(a_i)$ is the number of occurrences $a_i \in A$ in the multiset. As a rule, elements with $n_i = 0$ are omitted in formal sum. Union and subtraction of two multisets $\mu_1 = n_1a_1 + n_2a_2 + \ldots + n_ka_k$ and $\mu_2 = m_1a_1 + m_2a_2 + \ldots + m_ka_k$ on set A are defined accordingly as $\mu_1 + \mu_2 = (n_1 + m_1)a_1 + (n_2 + m_2)a_2 + \ldots + (n_k + m_k)a_k$ and $\mu_1 - \mu_2 = (n_1 - m_1)a_1 + (n_2 - m_2)a_2 + \ldots + (n_k - m_k)a_k$, 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, then this

multiset is denoted as $\mathbf{0}$, while empty set is denoted as a $\emptyset$. 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 $\mathcal{M}(A)$.

The finite *sequence* s on a set A is defined as function $s : \mathbb{N}_0 \to A \cup \emptyset$ that associates with non-negative integer number one element of the set A or element $\emptyset$ if number is greater than sequence size. Sequences are written as $(a_i)_{i=0}^n$ or more briefly (a_i). The set of all finite sequences on set A is denoted as (A). Element b belongs to sequence s, i.e. $b \in s$, if and only if $\exists i \in \mathbb{N}_0 \Longrightarrow s(i) = b$. We will denote $(a_i)_{i=0}^n \subseteq (b_j)_{j=0}^m$ that sequence a_i includes in sequence b_i if $m \geq n$ and $i, j \in [0 \dots n] : a_i = b_j$. When there is no ambiguity we will also denote an empty sequence as $\emptyset$.

3.2 Names Tree

In order to deal with OOP object names that form quite complicated name spaces, we introduce an auxiliary mathematical object – *names tree*. To avoid abundant repetitions we shall think that there are two universal alphabets: $\mathcal{A}$, a universal alphabet for names and Δ, an alphabet for labeling functions. To designate empty symbol, that $\notin \mathcal{A}$ and $\notin \Delta$, the symbol of empty set $\emptyset$ is used.

Definition 1. Let us define names tree as a tuple $\Psi = \langle v_0, \mathcal{V}, \mathcal{E}, nm \rangle$, where $\mathcal{V}$ is a set of nodes, $v_0 \in \mathcal{V}$ is a root node; $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of edges such, that

$$\forall v' \in \mathcal{V}, v' \neq v_0 \Longrightarrow \exists! path(v') \equiv (e_k)_{k=0}^n, e_k = (v_k, v_{k+1}) \in \mathcal{E}, v_{n+1} = v',$$

$nm : \mathcal{V} \to \mathcal{A}$ – a function returning node name.

Let us also associate with names tree Ψ a function $np : \mathcal{V} \to (\mathcal{A})$ returning named path from root:

$$np(v) = \begin{cases} (nm(v_{k+1}) \mid (v_k, v_{k+1}) \in path(v))_{k=0}^n, & \text{if } v \neq v_0, \\ \emptyset, & \text{if } v = v_0. \end{cases}$$

When $\exists! v \in \mathcal{V} : (v_0, v) \in \mathcal{E}$, then the tree can be referred to as a single-trunk names tree.

Briefly, names tree is defined as a directed rooted graph, with each node having a name from the universal alphabet $\mathcal{A}$ and a special function returning sequence of node names in the path from the root.

Names trees have several operations:

- formal union,
- step growth,
- and single-trunk rename.

Examples of these operations are shown in the Figure 2.

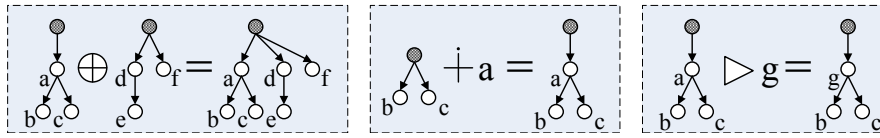

Figure 2. Examples of operations on names trees

Definition 2 (Formal union of names trees)**.** Given: Two names trees – Ψ_1 and Ψ_2. $\Psi_1 = \langle v_{01}, \mathcal{V}_1, \mathcal{E}_1, nm_1, np_1\rangle$, and $\Psi_2 = \langle v_{02}, \mathcal{V}_2, \mathcal{E}_2, nm_2, np_2\rangle$. The formal union of names trees Ψ_1 and Ψ_2 is the tree $\Psi = \Psi_1 \oplus \Psi_2 = \langle v_0, \mathcal{V}, \mathcal{E}, nm, np\rangle$, where v_0 the new root node,

$$\mathcal{V} = \mathcal{V}_1 \setminus \{v_{01}\} \cup \mathcal{V}_2 \setminus \{v_{02}\} \cup \{v_0\},$$

$$\mathcal{E} = \{(v', v'') \mid (v', v'') \in \mathcal{E}_1 \wedge v' \neq v_{01}\} \cup \{(v', v'') \mid (v', v'') \in \mathcal{E}_2 \wedge v' \neq v_{02}\}$$
$$\cup \{(v_0, v'') \mid (v_{01}, v'') \in \mathcal{E}_1\} \cup \{(v_0, v'') \mid (v_{02}, v'') \in \mathcal{E}_2\},$$

$$nm(v) = \begin{cases} \emptyset, & v = v_0, \\ nm_1(v), & v \in \mathcal{V}_1, \\ nm_2(v), & v \in \mathcal{V}_2. \end{cases}$$

By definition, the formal union of names trees operation is commutative and associative. Thus for a set of names trees $\Upsilon = \{\Psi_1, \Psi_2 \ldots \Psi_n\}$ the formal union operation can be in any order and we can write $\Psi_1 \oplus \Psi_2 \oplus \ldots \oplus \Psi_n \equiv \biguplus_{\Psi_i \in \Upsilon} \Psi_i \equiv \biguplus \Upsilon$.

Definition 3 (Step growth of names tree)**.** Given: Names tree $\Psi_1 = \langle v_{01}, \mathcal{V}_1, \mathcal{E}_1, nm_1, np_1\rangle$ and the name $q \in \mathcal{A}$. Operation of step growth of the names tree builds a new tree $\Psi = \Psi_1 \dot{+} q = \langle v_0, \mathcal{V}, \mathcal{E}, nm, np\rangle$, where v_0 – new root node, $\mathcal{V} = \mathcal{V}_1 \cup \{v_0\}$, $\mathcal{E} = \mathcal{E}_1 \cup \{(v_0, v_{01})\}$,

$$nm(v) = \begin{cases} nm_1(v), & v \in \mathcal{V}_1 \setminus \{v_{01}\}, \\ q, & v = v_{01}. \end{cases}$$

Formal union of names trees combines two or more trees into one by joining their root nodes, while the step growth of names tree operation gives a new name to the initial root node and adds a new unnamed root to the resulting tree.

Definition 4 (Renaming a single-trunk names tree)**.** Given: A single-trunk names tree $\Psi_1 = \langle v_0, \mathcal{V}, \mathcal{E}, nm_1, np_1\rangle$, having $\exists! v_1 \in \mathcal{V} : (v_0, v_1) \in \mathcal{E}$ and the name $q \in \mathcal{A}$. Renaming this single-trunk names tree Ψ_1 operation builds a new tree $\Psi = \Psi_1 \triangleright q = \langle v_0, \mathcal{V}, \mathcal{E}, nm, np\rangle$, where

$$nm(v) = \begin{cases} nm_1(v), & v \in \mathcal{V}_1 \setminus \{v_1\}, \\ q, & v = v_1. \end{cases}$$

4 PETRI NET OBJECTS AND OPERATIONS ON THEM

Using names tree introduced above, we define Petri net structured objects and operations on them to construct a model of an object-oriented program.

Definition 5. A Petri net is defined as tuple $\Sigma = \langle S, T, {}^\bullet(), ()^\bullet \rangle$, where S – finite set of places. T – finite set of transitions, with $S \cap T = \emptyset$. ${}^\bullet() : T \to \mathcal{M}(S)$ – incoming incidence function, $()^\bullet : T \to \mathcal{M}(S)$ – outgoing incidence function. Multisets of places ${}^\bullet t$ and $t^\bullet$ are referred to as incoming and outgoing multisets of transition $t \in T$ accordingly. Denotations ${}^\bullet t_\Sigma$ and $t^\bullet_\Sigma$ can be used to show the Petri net scope.

Definition 6. A Petri net structured object (PNS-object or just object, for short) is a tuple $E = \langle \Sigma, \Psi, \Gamma, M_0 \rangle$, where $\Sigma = \langle S, T, {}^\bullet(), ()^\bullet \rangle$ – Petri net, determining object behavior; $\Psi = \langle v_0, \mathcal{V}, \mathcal{E}, nm, np \rangle$ – a names tree representing structural design of a PNS-object; $M_0 \in \mathcal{M}(S)$ – initial marking; $\Gamma = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ – a set of access points (AP), each $\alpha_i = \langle id_i, in_i, out_i, \sigma_i \rangle$, where

- $id_i \equiv id(\alpha_i)$ – access point identifier,
- $in_i \equiv in(\alpha_i) \in \mathcal{V} \cup \emptyset$ – input role of access point α_i as position in names tree,
- $out_i \equiv out(\alpha_i) \in (\mathcal{A}) \cup \emptyset$ – output role of access point α_i as a named path in names tree,
- $\sigma_i \equiv \sigma(\alpha_i) : T \to \mathcal{M}(\Delta)$ – transitions labeling function.

The subset $In(\Gamma) = \{\alpha \in \Gamma \mid in(\alpha) \neq \emptyset \wedge out(\alpha) = \emptyset\}$ of access points is called object E input interface, and $Out(\Gamma) = \{\alpha \in \Gamma \mid in(\alpha) = \emptyset \wedge out(\alpha) \neq \emptyset\}$ – output interface.

The transition t label in the access point α is denoted for brevity as $\alpha_\Sigma(t) \equiv \sigma_\alpha(t)$ or just $\alpha(t)$. The labeling function σ is naturally extended to the multiset of transitions $\sigma : \mathcal{M}(T) \to \mathcal{M}(\Delta)$ by the next way: $\forall \Theta = \Sigma n_i t_i \in \mathcal{M}(T) \Longrightarrow \sigma(\Theta) = \Sigma n_i \sigma(t_i)$.

Less formally, PNS-object is a Petri net with a set of labeling functions which are accessible via names and nodes in names tree.

Now we define a number of operations on objects that will be used later in the construction of the program model.

Definition 7 (Union of access points)**.** Given: A PNS-object $E_1 = \langle \Sigma_1, \Psi_1, \Gamma_1, M_{01} \rangle$, that has two access points $\alpha, \beta \in \Gamma_1$) with identical input and output identifiers: $in(\alpha) = in(\beta), out(\alpha) = out(\beta)$. Operation uniting access points α and β of E_1 builds a new PNS-object $E = (E_1)_{\gamma=\alpha+\beta} = \langle \Sigma_1, \Psi_1, \Gamma, M_{01} \rangle$, so that $\Gamma = \Gamma_1 \backslash \{\alpha, \beta\} \cup \{\gamma\}$ where $\gamma = \langle \langle id(\alpha) id(\beta) \rangle, in_\alpha, out_\alpha, \sigma_\gamma = \sigma_\alpha + \sigma_\beta \rangle$.

Access point union operation, instead of two initial access points, creates a new access point that combines their labeling functions. This operation is required, for example, when control flow models having calls to the same method or function are united. By definition the union of access points operation is associative and transitive.

Definition 8 (Formal union of PNS-objects)**.** Suppose we are given two objects E_1 and E_2. $E_1 = \langle \Sigma_1, \Psi_1, \Gamma_1, M_{01} \rangle$, $E_2 = \langle \Sigma_2, \Psi_2, \Gamma_2, M_{02} \rangle$, where $\Sigma_1 = \langle S_1, T_1, {}^\bullet()_1, ()_1^\bullet \rangle$ and $\Sigma_2 = \langle S_2, T_2, {}^\bullet()_2, ()_2^\bullet \rangle$. The result of PNS-objects E_1 and E_2 formal union is the object $E = E_1 \oplus E_2 = \langle \Sigma, \Psi, \Gamma, M_0 \rangle$, where $\Sigma = \langle S, T, {}^\bullet(), ()^\bullet \rangle$,

$$S = S_1 \cup S_2, \quad T = T_1 \cup T_2, \quad M_0 = M_{01} + M_{02},$$

$${}^\bullet() = {}^\bullet()_1 \cup {}^\bullet()_2, \quad ()^\bullet = ()_1^\bullet \cup ()_2^\bullet, \quad \Gamma = \Gamma_1 \cup \Gamma_2, \quad \Psi = \Psi_1 \oplus \Psi_2.$$

By definition PNS-objects formal union operation is commutative and associative. So, for a set of PNS-objects $G = \{E_1, E_2, \ldots, E_n\}$, the formal union operation can be done in any order and we can write $E_1 \oplus E_2 \oplus \ldots \oplus E_n \equiv \biguplus_{E_i \in G} E_i \equiv \biguplus G$.

Definition 9 (Restriction by access point)**.** Consider PNS-object $E_1 = \langle \Sigma_1, \Psi_1, \Gamma_1, M_0 \rangle$, where $\Sigma_1 = \langle S_1, T_1, {}^\bullet()_1, ()_1^\bullet \rangle$ and $\Psi_1 = \langle v_{01}, \mathcal{V}_1, \mathcal{E}_1, nm_1, np_1 \rangle$. Restriction of object E_1 by access point $\alpha \in \Gamma_1$ forms a new object $E = \partial_\alpha(E_1) = \langle \Sigma, \Psi_1, \Gamma, M_0 \rangle$, where $\Sigma = \langle S_1, T, {}^\bullet(), ()^\bullet \rangle$ and $T = T_1 \backslash \{t \in T | \alpha(t) > \mathbf{0}\}$, $\forall t \in T | {}^\bullet(t) = {}^\bullet(t)_1, (t)^\bullet = (t)_1^\bullet$, $\Gamma = \Gamma_1 \backslash \{\alpha\}$.

Restriction of a PNS-object by an access point deletes each transition labeled nonempty by the access point with all adjacent arcs. The restriction operation is associative: $\partial_{\alpha_2}(\partial_{\alpha_1}(E)) = \partial_{\alpha_1}(\partial_{\alpha_2}(E))$ and can be simply generalized to a subset of access points used for restriction. $E = \partial_{\alpha_n}(\ldots \partial_{\alpha_2}(\partial_{\alpha_1}(E_1))) \equiv \partial_U(E_1)$, where $U = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$.

Definition 10 (PNS-object normalization)**.** For the given object $E_1 = \langle \Sigma_1, \Psi_1, \Gamma_1, M_{01} \rangle$, $\Sigma = \langle S_1, T_1, {}^\bullet()_1, ()_1^\bullet \rangle$ we can construct the normalized PNS-object $E = norm(E_1)$, where $E = \langle \Sigma, \Psi, \Gamma, M_0 \rangle$, $\Sigma = \langle S, T, {}^\bullet(), ()^\bullet \rangle$ and:

- $S = S_1 \setminus \{s_i \mid \forall t \in T_1 : s_i \notin {}^\bullet(t)_1 \wedge s_i \notin (t)_1^\bullet\}$,
- $T = T_1 \setminus \{t_i \mid {}^\bullet(t_i)_1 = \mathbf{0} \wedge (t_i)_1^\bullet = \mathbf{0}\}$,
- $\Gamma = \{\alpha | \exists \alpha' \in \Gamma' : in(\alpha) = in(\alpha'), out(\alpha) = out(\alpha')\}$, such that
  - $\forall \alpha' \in \Gamma' \Longrightarrow \exists! \alpha \in \Gamma : in(\alpha) = in(\alpha'), out(\alpha) = out(\alpha')$
  - $\forall \gamma \in \Gamma \Longrightarrow \sigma_\gamma = \sum_{\alpha' \in \Gamma' : in(\alpha) = in(\gamma), out(\alpha) = out(\gamma)} \sigma_\alpha$.

Normalization removes unlinked places or transitions and unites access points with the same roles. It is assumed, that all following objects being addressed are either normalized or that a normalization operation can be applied to them.

Definition 11 (Simple composition)**.** Given: A PNS-object $E = \langle \Sigma_1, \Psi, \Gamma, M_0 \rangle$ and its access points $\alpha, \beta \in \Gamma_1$, where $\Sigma_1 = \langle S_1, T_1, {}^\bullet()_1, ()_1^\bullet \rangle$. Operation of simple composition by access points α and β of PNS-object E_1 forms a new object $E = \langle \Sigma, \Psi, \Gamma, M_0 \rangle$, $\Sigma = \langle S_1, T, {}^\bullet(), ()^\bullet \rangle$ where

1. $T = T_1 \cup T_{syn}$, where $T_{syn} = \{\mu_1 + \mu_2 | \mu_1, \mu_2 \in \mathcal{M}(T_1), \sigma_\alpha(\mu_1) = \sigma_\beta(\mu_2) > \mathbf{0}$, sum $\mu_1 + \mu_2$ is minimal, i.e. no sum $\mu_1' + \mu_2'$ exists that $\mu_1' + \mu_2' < \mu_1 + \mu_2$ and $\sigma_\alpha(\mu_1') = \sigma_\beta(\mu_2')\}$,

2. ${}^\bullet() = {}^\bullet()_1 \cup \{({}^\bullet(\mu_1)_1 + {}^\bullet(\mu_2)_1, \mu_1 + \mu_2) | \mu_1 + \mu_2 \in \mathcal{M}(T), \mu_1, \mu_2 \in \mathcal{M}(T_1)\}$,
3. $()^\bullet = ()^\bullet_1 \cup \{(\mu_1 + \mu_2, (\mu_1)^\bullet_1 + (\mu_2)^\bullet_1) | \mu_1 + \mu_2 \in \mathcal{M}(T), \mu_1, \mu_2 \in \mathcal{M}(T_1)\}$,
4. $\Gamma = \Gamma_1 \cup \{\gamma\}$, where $\gamma = \langle id_\gamma, in(\alpha), \emptyset, \sigma_\gamma \rangle$ and $id_\gamma = \langle id(\alpha) id(\beta) \rangle$,
5. $\forall t \in T_1, \xi \in \Gamma \setminus \{\gamma\} : \xi_\Sigma(t) = \xi_{\Sigma_1}(t), \gamma(t) = \mathbf{0}$,
6. $\forall t = (\mu_1 + \mu_2) \in T_{syn}, \forall \xi \in \Gamma \setminus \{\alpha, \beta, \gamma\} : \xi(t) = \xi(\mu_1) + \xi(\mu_2)$, $\sigma_\gamma(t) = \alpha(\mu_1)$, $\sigma_\alpha(t) = \mathbf{0}$, $\sigma_\beta(t) = \mathbf{0}$.

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

$$E = [E_1]^\alpha_\beta, \qquad E = E_1 \ {}_\alpha[\ ]_\beta \ E_2 \equiv [E_1 \oplus E_2]^\alpha_\beta.$$

Operation of simple composition adds to object E_1 a number of new synchronization transitions T_{syn}. New transitions are defined by multisets of symbols $\mu_1 + \mu_2$, where $\mu_1, \mu_2 \in \mathcal{M}(T)$. Incoming and outgoing multisets of the new transitions are calculated from constituting multisets accordingly: ${}^\bullet(\mu_1 + \mu_2) = {}^\bullet(\mu_1) + {}^\bullet(\mu_1), (\mu_1 + \mu_2)^\bullet = (\mu_1)^\bullet + (\mu_2)^\bullet$. Initial transitions preserve original labeling. Labeling for new transitions is calculated from constituting multisets for all access points except those used in operation: $\sigma(t) = \sigma(\mu_1) + \sigma(\mu_2)$. Operational access points have empty labeling in new transitions. And one new access point is added that designates synchronization with labeling used for multiset calculation: $\sigma_\gamma(t) = \sigma_\alpha(\mu_1) = \sigma_\beta(\mu_2)$.

Definition 12 (Structural composition (encapsulation))**.** Given: A PNS-object $E_1 = \langle \Sigma_1, \Psi_1, \Gamma_1, M_0 \rangle$, $\Psi_1 = \langle v_{01}, \mathcal{V}_1, \mathcal{E}_1, nm_1, np_1 \rangle$ and a name $q \in \mathcal{A}$. The unary form of structural composition operation constructs from the initial PNS-object E_1, a new object $E \equiv \succ(q, E_1) = \langle \Sigma, \Psi, \Gamma, M_0 \rangle$ in the following steps:

1. Build the intermediate set of access point pairs $SYNC = \{\langle \alpha_i, \beta_i \rangle | i \in [1 \dots n], \alpha_i \in In(\Gamma_1), \beta_i \in Out(\Gamma_1), np_1(in(\alpha_i)) = out(\beta_i)\}$.
2. Build the access points set for further restriction $W = \{\beta_i \in Out(\Gamma_1) | \exists \langle \alpha_i, \beta_i \rangle \in SYNC\}$.
3. Make the simple composition for all intermediate access point pairs $\forall i \in [1 \dots n] \Longrightarrow E_{i+1} = [E_i]^{\alpha_i}_{\beta_i}, \langle \alpha_i, \beta_i \rangle \in SYNC$.
4. Restrict the PNS-object by prepared access points set $E' = \partial_W(E_{n+1}) = \langle \Sigma, \Psi', \Gamma, M_0 \rangle$.
5. Add the new root $\Psi = \Psi' \dotplus q$.

For more than one object $X = \{E_1, \dots, E_n\}$, the operation of structural composition is defined as

$$E = \succ(q, E_1, \dots, E_n) \equiv \succ(q, (E_1 \oplus \dots \oplus E_n)) \equiv \succ(q, X).$$

This definition implies that a unary structural composition performs a set of simple composition operations on original object that are defined by matching pairs of incoming access point identifier paths and outgoing access point names. Then it deletes transitions of satisfied outgoing access points and finally regroups incoming access points into a new structure.

Definition 13 (Absorption operation)**.** Given: Two PNS-objects $E' = \langle \Sigma', \Psi', \Gamma', M_0' \rangle$, $\Psi' = \langle v_0', \mathcal{V}', \mathcal{E}', nm', np' \rangle$ and $E'' = \langle \Sigma'', \Psi'', \Gamma'', M_0'' \rangle$, $\Psi'' = \langle v_0'', \mathcal{V}'', \mathcal{E}'', nm'', np'' \rangle$. Absorption operation of PNS-object E'' by PNS-object E' constructs a new object $E \equiv \boxplus(E', E'') = \langle \Sigma, \Psi, \Gamma, M_0 \rangle$ in the following steps:

1. Build the intermediate set of access point pairs $SYNC = \{\langle \alpha_i, \beta_i \rangle | i \in [1 \dots n], \alpha_i \in In(\Gamma_2), \beta_i \in Out(\Gamma_1), np_2(in(\alpha_i)) = out(\beta_i)\}$.
2. Build the access points set for further restriction $W = \{\beta_i \in Out(\Gamma_1) | \exists \langle \alpha_i, \beta_i \rangle \in SYNC\}$.
3. Formally units given objects $E_1 = E' \oplus E''$.
4. Make the simple composition for all intermediate access point pairs $\forall i \in [1 \dots n] \Longrightarrow E_{i+1} = [E_i]_{\beta_i}^{\alpha_i}, \langle \alpha_i, \beta_i \rangle \in SYNC$.
5. Restrict the PNS-object by prepared access points and by $In(\Gamma_2)E = \partial_{In(\Gamma_2)} (\partial_W(E_{n+1}) = \langle \Sigma, \Psi, \Gamma, M_0 \rangle)$.

For more than one absorbed object $X = \{E_1, \dots, E_n\}$, operation of structural composition is defined as

$$E = \boxplus(E', E_1, \dots, E_n) \equiv \boxplus(E', (E_1 \oplus \dots \oplus E_n)) \equiv \boxplus(q, X).$$

An absorption operation executes all compositions between host object and its "guests". Then "guest" objects are restricted by their input interfaces so that their further synchronizations are possible only via output interface.

5 GRAPHICAL NOTATION

Graphical representation of PNS-objects and all rules necessary to draw operations on them are shown in Figure 3. Each rule is marked by callout with an ordinal number.

1. PNS-objects are drawn in rectangles bounding the area of objects contents. The object name is placed inside the rectangle. Comments about the object are placed in parentheses after to the name.
2. Inside the PNS-object a Petri net or a composition of other PNS-objects can be drawn. Petri nets are drawn using standard graphical notation as a bipartite directed graph.
3. PNS-object incoming and outgoing access points sets are represented by isosceles triangles placed on rectangle borders. For incoming access points the base of

triangles is drawn outside the object bounding rectangle. For outgoing access point – vice versa.

4. Link connecting access point sets represents operation of objects simple composition. A line over a triangle vertex opposite the base depicts the restriction by an access point operation that is done by a certain set of access points in this case by access points performing the simple composition operation.
5. A double border rectangle is used for structured PNS-object composition and absorption operations. A single border rectangle represents a formal PNS-objects union operation.
6. A thick border rectangle around a PNS-object adjoining a double border rectangle represents a net performing absorption operation.

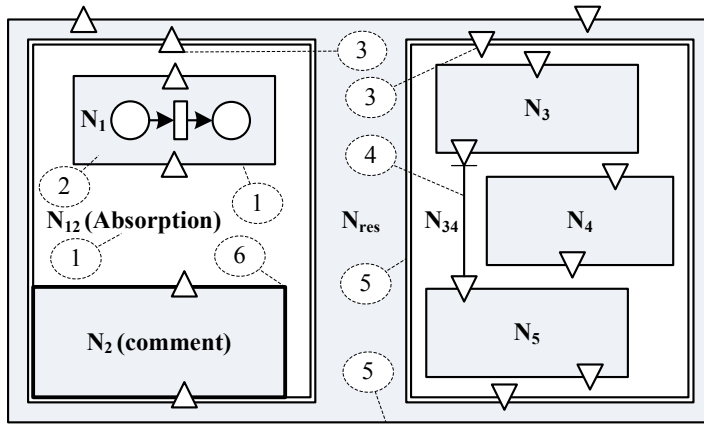

Figure 3. Graphical representation of PNS-objects and composition operations

6 MODELING OOP ELEMENTS

We identify OOP objects (methods, members, classes, functions and global variables) from some alphabet $\mathcal{A}$ and we do not distinguish the objects from their identifiers. The Δ alphabet defines "visible" actions of program behaviour. In this article it is considered to consist of just two elements $\Delta = \{bgn, end\}$, that represent actions of function start and function finish. There are some more actions that may be of interest in modeling program behaviour such as reading from and writing to variables and others.

Definition 14 (Model of a class method or global function declaration)**.**
A PNS-object $E \equiv Dcl(f) = \langle \Sigma, \Psi, \Gamma, M_0 \rangle$ is called a model of a function or a class method f declaration, if and only if

1. $\Sigma = \langle S = \{s_0\}, T = \{t_1, t_2\}, ()^\bullet = \{(t_1, s_0)\}, {}^\bullet() = \{(t_2, s_0)\}\rangle$;
2. $\Psi = \langle v_{0f}, \mathcal{V} = \{v_{0f}, v_1\}, \mathcal{E} = \{(v_{0f}, v_1)\}, nm(v_1) = f, np(v_1) = f\rangle$;
3. $\Gamma = \{\alpha_{in}, \alpha_{out}\}$, $\alpha_{in} = \langle id_{in}, v_1, \emptyset, \sigma_i\rangle$, $\alpha_{out} = \langle id_{out}, \emptyset, out = (f), \sigma_o\rangle$ where $\sigma_i = \sigma_o = \{(t_1, bgn), (t_2, end)\}$;
4. $M_0 = \mathbf{0}$.

Less formally, a model of a function or method declaration is unambiguously built from two transitions and one place connected with each other and properly labeled. For any method or function f a model of its declaration is denoted as $Dcl(f)$. We consider models of class methods and functions in OOP in the same manner because their differences are revealed in composition of objects.

The next two definitions about classes and class members (or objects) are recursive and support each other. This is similar to common object-oriented programming languages where class members can be built from other classes.

Definition 15 (Model of a class declaration)**.** For class C having a set of methods $Meth = \{F_i\}$ and a set of members $Memb = \{D_j\}$ the PNS-object E is called a model of the class declaration if and only if

$$E = \nearrow(C, Dcl(Meth) \oplus Dcl(Memb))$$

where $Dcl(Meth) = \biguplus Dcl(F_i)$ is the formal union of the method declaration models, and $Dcl(Memb) = \biguplus Dcl(D_j)$ is the formal union of the member declaration models.

In other words, class declaration is the structural composition of all its method and member declarations. A class declaration has a single-trunk names tree as it follows from the definition of structural composition operation. For classes without members, the model of its declaration can be built from the above definitions. We must now define members.

Definition 16 (Model of an object declaration)**.** For a global variable or a class member D that is an instance of a class C with a declaration model $Dcl(C) = \langle \Sigma, \Psi, \Gamma, M_0\rangle$ the PNS-object E is called a model of D declaration if it is built as

$$E \equiv Dcl(D) = \langle \Sigma, \Psi \triangleright v, \Gamma, M_0\rangle.$$

An object or a member declaration model is just a copy of its class declaration with a renamed single-trunk names tree representing the member structural design. Members that are not class instances are not considered in this paper though we can model read and write operations on them as functions.

Definition 17 (Model of a class method or global function implementation)**.**
Given: A function or a class method f with a set of its variables $Var = \{D_i\}$ and its control flow model as a PNS-object $E_f = \langle \Sigma_f, \Psi_f, \Gamma_f, M_{f0}\rangle$. The PNS-object E

is called a model of the function or the class method f implementation, if it is built as:

$$E \equiv Imp(f) = \boxplus(E_f, Dcl(Var))$$

where $Dcl(Var) = \biguplus Dcl(D_i)$, the formal union of the variables declaration models.

The model of a class method (or function) implementation is an absorption of its local variables declaration models by the method's (or function's) control flow model. Examples of function declarations and implementations are shown in the next Section. Control flow creation is not discussed here as it can be accomplished automatically by syntax directed translation, and for our needs function calls are the only constructions that must have labeling.

Definition 18 (Model of a class implementation)**.** For a class C having a set of methods $Meth = \{F_i\}$ and a set of members $Memb = \{D_j\}$ the PNS-object E is called a model of the class implementation if and only if

$$E \equiv Imp(C) = \succ(C, Imp(Meth) \oplus Dcl(Memb))$$

where $Imp(Meth) = \biguplus Imp(F_i)$ is the formal union of the class method implementation models and $Dcl(Memb) = \biguplus Dcl(D_j)$ is the formal union of the class member declaration models.

In the class implementation model, methods are presented by their implementation models and members by declaration models. This makes modeling methods uniform for all objects in the program. Variables are also included through model declaration that further will be synchronized with class model.

Definition 19 (Model of a module Q)**.** For a module Q having a set of functions $F = \{F_i\}$, a set of classes $C = \{C_i\}$ and a set of global objects $V = \{V_i\}$ the PNS-object E is called a model of the Q module, if and only if

$$E \equiv Imp(Q) = Imp(F) \oplus Imp(C) \oplus Dcl(V)$$

where $Imp(F) = \biguplus Imp(F_i)$ is the formal union of the function implementation models, $Imp(C) = \biguplus Imp(C_i)$ is the formal union of the class implementation models, and $Dcl(V) = \biguplus Dcl(V_i)$ is the set of models of the global object declarations.

A module is considered as a translation unit in common object-oriented programming languages (i.e. one source file) which may group classes, functions, and global variables. In the same way, a module model can consist of at least one function or variable or class. Module models can be united by PNS-objects formal union operation.

Definition 20 (A full model of an OO program)**.** Given: A PNS-object *Loader* and a module $Q = \langle \Sigma_Q, \Psi_Q, \Gamma_Q, M_{Q0} \rangle$ containing all program functions, variables and classes with $\Psi_Q = \langle v_0, \mathcal{V}, \mathcal{E}, nm, np \rangle$. A PNS-object

$$E = \langle \Sigma, \Psi, \Gamma, M_0 \rangle \equiv \boxplus(Loader, Imp(Q))$$

is a full model of an OO program if and only if E has an empty input interface, i.e. $\forall \alpha \in \Gamma : out(alpha) = \emptyset$.

The definition of a full program model supposes that all output interfaces are resolved by composition operations. After absorption, there will be no input interfaces because there is no further need of them. In practice we cannot resolve all function calls in a program model because building a program does not require source texts of libraries. Having them would require source texts of system functions and so on. The only way to obtain a full program model is to use human-made models of all necessary library functions. Another way is to use an incomplete program model resulting from an absorption operation of the program module by PNS-object *Loader*.

The object *Loader* introduced here models the start of the program; we assume that the model of *Loader* is defined and consists only from the call of the function `main`. The other *Loader* roles, such as the variable initialization before and destruction after function roles such as variables initialization before and destruction after function `main`, are not considered in this article.

Now we introduce an algorithm of object-oriented program model construction in terms of structured Petri net objects. Assume that we have a translator from an object-oriented programming language into some tree-like representation further referred to as a parse tree. Tree nodes have *names*, *types*, *roles* and contents. There are quite a lot of restrictions on the programs considered. *Type* of node in the program parse tree represents one of several possibilities: global namespace, class, method, function, variable and member. Nodes have either a declaration or implementation *role*. We specify that the program has no class inheritance, no polymorphism and no static members. The program must compile successfully thereby removing all the syntax errors and unresolved name problems. Algorithm 1, presented below, forms an OOP model in terms of PNS-objects. The overall algorithm scheme is built in three stages. First a recursive procedure generates PNS-objects for all declarations in the program and for all function and method implementations. As far as a class implementation is spread over its members implementations, this stage also prepares content of class implementation PNS-objects. In the second stage operation of structured composition makes class implementation models from prepared contents. In the third stage, all generated PNS-objects are combined into the program model by an absorption operation using a model of the program loader.

7 AN EXAMPLE OF OBJECT-ORIENTED C++ PROGRAM MODEL

Let us briefly describe the process of a C++ program model construction in terms of structured Petri net objects on the "set division" problem example. The program considered consists of three translation units. The first is `divproc.cpp`, with two included headers `divproc.h` and `set.h`. The second unit is `main.cpp` file that includes headers `set.h`, and `divproc.h`. And the third is `set.cpp` file with the header `set.h`. Other header files listed in the source code are not questioned as

Algorithm 1 OOP model construction in terms of PNS-objects

Input: *ParseTree* is a tree of program units.
Output: PNS-object with empty input and output interface

```
1: function Translator(ParseTree)
2:     Model ← CreateNetForGobalNamespace()
3:     List ← CreateList()
4:     for all node ∈ ParseTree do
5:         ProcessUnit(node, Model, Model, List)
6:     for all net ∈ Model ∧ net ∈ ClassImplementation do
7:         StructuredComposition(net.Name, net)
8:     Loader ← GenerateLoader()
9:     AbsorptionComposition(Loader, Model)
10:    return Model
11: function ProcessUnit(Unit, Snet, Gnet, CDList)
12:    switch type of Unit do
13:        case ClassDeclaration:
14:            CDnet ← CreatePNSObject(Unit)                ▷ assigns name to net
15:            for all node ∈ Unit do          ▷ Formal Union of content units models
16:                ProcessUnit(node, CDnet, Gnet)
17:            CDList << StructuredComposition(CDnet.Name, CDnet)
18:        case MethodImplementation:
19:            MInet ← CreatePNSObject(Unit)
20:            for all node ∈ Unit do                ▷ Formal Union of Var. declarations
21:                ProcessUnit(node, MInet, Gnet)
22:            CFnet ← CreateControlFlowNet(Unit)
23:            AbsorptionComposition(CFnet, MInet)
24:            CInet ← GetOrCreateClassImplementationNet(Unit, Gnet)
25:            CInet << CFnet
26:        case FunctionImplementation:
27:            FInet ← CreatePNSObject(Unit)
28:            for all node ∈ Unit do                ▷ Formal Union of Var. declarations
29:                ProcessUnit(node, FInet, Gnet)
30:            CFnet ← CreateControlFlowNet(Unit)
31:            Gnet << AbsorptionComposition(CFnet, FInet)
32:        case MethodDeclaration:
33:            Snet << GenerateMethodDeclaration(Unit)
34:        case FunctionDeclaration:
35:            Snet << GenerateFunctionDeclaration(Unit)
36:        case MemberDeclaration:
37:            MDnet ← GenerateMemberDeclaration(Unit, CDList)
38:            Snet << MDnet
39:            CInet ← GetOrCreateClassImplementationNet(Unit, Gnet)
40:            CInet << MDnet
41:        case VariableDeclaration:
42:            Snet << GenerateVariableDeclaration(Unit)
```

we accept all third-party library functions as correct and therefore irrelevant to our program execution.

The parse tree of the entire program is built by merging parse trees of all translations units. The merger is done by names and roles of the trees nodes which unite all occurrences of the same function declaration or of the same variable declaration. Removing `set.h` functions and `stdio.h` functions from consideration, the resulting parse tree has the following list of units on the first level from root: class `Set` declaration, class `DivProc` declaration, ten `Set` method implementations, eight `DivProc` method implementations and the function `main` implementation. According to Algorithm 1 these units are translated into two PNS-objects modeling class declarations, two PNS-objects modeling class implementations and the PNS-object of `main` function implementation.

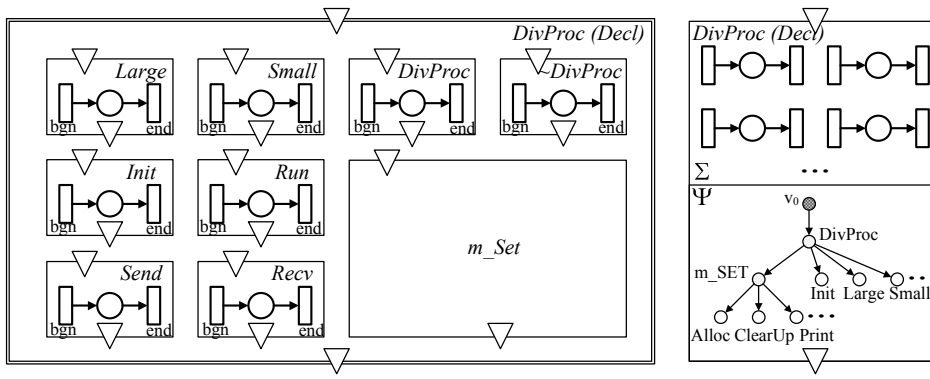

Figure 4. Model of `DivProc` class declaration in "DivSet" program example

Consider a model of the class `DivProc` declaration shown in the Figure 4. The left side of the figure depicts compositional representation of the declaration model made from declaration models of all members and methods. All Petri nets of methods in this composition are triples of begin transition, inner state and end transition. They differ by incoming and outgoing access points. Input access points of these objects are identified by their names in the program. Output access points are identified by the fully qualified names of the modeled methods in the program, including the name of the class. Input access point identifiers are transferred in the names tree after composition operations, while output access points identifiers remain constant thereby postponing actual binding of object method calls to the final stage of program model construction. The only member declaration model included in the `DivProc` class declaration is PNS-object modeling access to member `m_Set`. Double borders surrounding all inner objects depict encapsulation operations modifying access to members and methods so that their names are moved inside the class name scope. Non-compositional representation of the PNS-object and its names tree are shown in the right side of Figure 4.

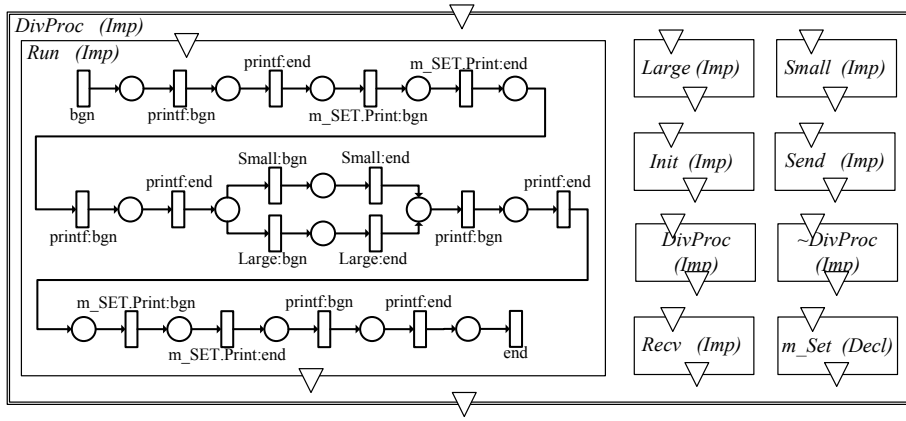

Figure 5. Model of DivProc class implementation in "DivSet" program example

Figure 5 shows a compositional model of `DivProc` class implementation. This model is obtained by encapsulation of all method implementation models and all member declaration models in class context. As an example, method `Run` is shown with a Petri net describing its control flow behavior. While the algorithm of this Petri net construction is not the topic of this article, we mention that it basically uses paired transitions to simulate control flow transfer between caller and callee functions and methods. So, the first transition simulates control flow entrance to the method `Run`. Then three pairs of transitions correspond to Line 8 of file `divproc.cpp` and model sequential calls to function `print` and method `Print` of object `m_SET`. Then two pairs of transition model choice between calls to method `Small` and method `Large` in lines 9–10. Then again, three pairs of transitions for line 11 are exactly the same like for line 8. The final transition simulates exit from the method.

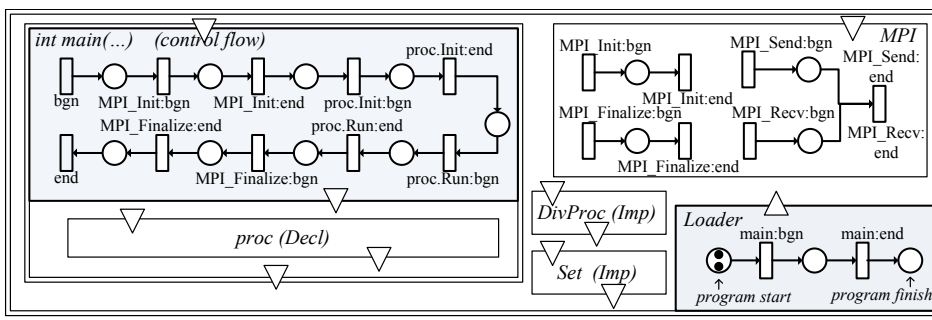

Figure 6. Model of "DivSet" program

The input interface in the method's PNS-object consists of a single access point

and the output interface of the three access points. The input access point labels the outermost transitions of the PNS-object. Output access points label transition pairs corresponding to function and method calls. The method `Run` has no internal variables and no absorption operation was shown.

The full "set division" program model is produced by the absorption operation shown in the Figure 6 consisting of five PNS-objects. Two PNS-objects represent `DivProc` and `Set` class implementation models. Also there is the function `main` implementation model, where the control flow model, shown with a Petri net, absorbs the variable `proc` declaration model. Our model requires an MPI model to simulate process interaction; it is presented as a PNS-object that simplistically models `MPI_Init`, `MPI_Finalize`, `MPI_Send` and `MPI_Recv` functions for the case of two processes in the "set division" program. And finally the *Loader* object is an initialization code model for this object-oriented program. The *Loader* model is presented in the most simplified form, having only one pair of transitions for a call of function `main` and two tokens in the start place to initiate two processes (*Small* and *Large*). The resulting PNS-object simulates behaviour of the entire program in terms of *visible* function calls. This is a raw model that can be refined by adding data inscriptions or guarding against impossible transition firings. This object can be automatically constructed on the base of syntax directed parsing process, excepting system library models (MPI, etc.) that need to be prepared manually.

8 RELATED WORK

The problem of program modeling in terms of Petri nets has a long history. For example in 1981 Peterson [14] wrote: "Petri nets can best represent the control structure of programs. Petri nets are meant to model sequences of instructions and the flow of information and computation but not the actual information values themselves". He showed that program control flow can be represented as a *flowchart*, which may in turn be represented in the form of Petri nets. Reisig [15] formulated this idea more straightforwardly. Program instructions are depicted as events (transition), and possible program states as conditions (places). Program process states are marked by tokens and the movement of the tokens represents program execution. With an extension of Petri nets called Coloured Petri Nets [16] it became possible to use simple variables in program models in a way close to the real programs.

In the 1990's programming languages supporting object-oriented concepts became widely available. Petri nets were subjected to the influence of ideas quite like programming languages. Lakos [17] showed how a set of Petri Net extensions arose. The concept of Coloured Petri Nets was extended to Hierarchical Coloured Petri Nets (HCPNs) by introducing transitions substitution, then to Modular Coloured Petri Nets (MCPNs) by substitution of places. Object-Based Petri Nets (OBPNs) enhanced MCPNs by allowing tokens to be subnets encapsulating their own activity. Object-Oriented Petri Nets were derived from Object-Based Petri Nets by including the notion of inheritance together with the associated polymorphism and dynamic

binding. And finally Object Petri Nets were derived from Object-Oriented Petri Nets by inclusion of test and inhibitor arcs. A similar approach closer to programming languages was used in [18] where Object Coloured Petri Nets (OCP-Nets) were introduced as an extension of Coloured Petri Nets. An OCP-Net is a set of class nets which offers services to other class nets by encapsulating it within a pair of IN- and OUT-transition. Other class nets call a service via a pair of an INV- and REC-transition. This corresponds to methods and method calls in object oriented programming languages. Another way of dealing with object-oriented concepts was presented in [19], where $OB(PN)^2$ language was introduced that has syntax close to programming languages while its semantics were defined in terms of Petri Nets extension M-nets and operations on them.

Despite many extensions of Petri nets supporting object-oriented concepts, there are only a few works related to automated program model construction in terms of Petri nets. Voron and Kordon [8] proposed automation of program model construction with GCC to understand and simplify program sources mainly from C language. It produces program descriptions in terms of blocks and links where blocks are grouped to form function control flow graphs. In [7] Westergaard considered translation process as syntax-directed translation using templates for each programming language syntax construct. This work used simplified procedural programming language assuming that the approach could be adjusted for real programming languages by only rewriting the grammar rules. At the same time other authors [6] proposed a guideline for object-oriented program modeling with simple human involvement in model construction. Moreover, they supposed that no automation for a high level object oriented language, such as Python, was available.

In summary, there are no discrepancies in building pure control flow models, so that procedural program models in terms of Petri nets produced by different tools are not very different from each other. The object-oriented paradigm with its declarative nature of notions introduces a new challenge in program modelling. It is possible to integrate object-oriented concepts in modelling languages as shown above, but it does not result in new analytical techniques. In this article we tried to preserve the simplicity of multi-labeled Petri nets while describing object-oriented properties of programs.

9 CONCLUSION

This paper describes a method designed for automated construction of object-oriented program models in terms of multilabeled Petri nets. We focused on encapsulation – one of the three concepts of object-oriented paradigm. To model encapsulation we built a program model from models of its object-oriented components: objects, methods, members and functions. The variety of object-oriented components form a space of names that actually has a tree like form. To deal with this variety we introduced names tree formal objects that can unite and grow making complex structures from simple ones. We then introduce a Petri net structured

object with a set of incoming and outgoing access points. Each incoming access point can be considered as a representation of a method or function. Each outgoing access point refers to a method or function when modeling the calls. Employing PNS-objects created defined models of declarations and implementations of functions, variables, classes and methods. Each definition implies an algorithm to build the appropriate model, and we introduced the algorithm of the entire program model construction.

There are many questions closely related to OOP modeling remaining outside our article that require further research. The first question is to cover inheritance and polymorphism concepts of object oriented paradigm in the OOP model. The second is to introduce proper representations of dynamic object creation and object pointers. A third question relates to detailed modeling of the Loader which was designed to deal with global object initializations performed prior to the call of the function `main`.

Acknowledgements

This work was supported by the research program "Fundamental Problems of Mathematical Modeling" of the Presidium of the Russian Academy of Sciences (Project 0262-2015-0139) and state funding (Project 0262-2014-0003).

10 APPENDIX: SAMPLE PROGRAM SOURCE CODE

We use the following real parallel program to demonstrate essentials of compositional model construction in terms of structured Petri net objects. The program consists of five files: `set.h`, `set.cpp`, `divprocess.h`, `divprocess.cpp`, and `main.cpp`. Where `set.*` files contain class declaration and implementation for a set of integer numbers. `divprocess.*` files contain declaration and implementation of two MPI-processes for the "set division" problem, and `main.cpp` is an entry point to the program.

`set.h`

```
1  #ifndef __SET_H__
2  #define __SET_H__
3
4  class Set {
5  private: // Internal data
6    int count;
7    int *data;
8  private: // Internal behaviour
9    int *Alloc(int cnt) { return new int[cnt]; };
10   void ClearUp(void) { delete data; count = 0; };
11 public: // Constructor/destructor
12   Set() { data = 0; count = 0; };
```

```
13   ~Set() { ClearUp(); };
14 public: // Public methods
15   void InitCmdLine(int argc, char *argv[]);
16   void Print(void);
17   int min();
18   int max();
19   void operator+(int value);
20   void operator-(int value);
21 };
22
23 #endif
```

set.cpp

```
1  #include "set.h"
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  void Set::InitCmdLine(int argc, char *argv[]) {
6    count=argc-1; data=Alloc(count);
7    for (int i=1; i<argc; i++) data[i-1]=atoi(argv[i]);
8  }
9  void Set::Print(void) {
10   for (int i=0; i<count; i++) printf("%i ",data[i]);
11 }
12 int Set::min() {
13   int mn=data[0];
14   for (int i=1; i<count; i++) if (data[i]<mn) mn=data[i];
15   return mn;
16 }
17 int Set::max() {
18   int mx=data[0];
19   for (int i=1; i<count; i++) if (data[i]>mx) mx=data[i];
20   return mx;
21 }
22 void Set::operator+(int value) {
23   int *ptr=Alloc(count+1);
24   for (int i=0; i<count; i++) ptr[i]=data[i];
25   ptr[count++]=value; delete data; data=ptr;
26 }
27 void Set::operator-(int value) {
28   int *ptr=Alloc(count-1);
29   for (int i=0, j=0; i<count; i++, j++)
30     if (data[i]!=value) ptr[j]=data[i]; else j--;
```

```
31   count--; delete data; data=ptr;
32 }
```

divproc.h

```
1  #ifndef __DIVPROC_H__
2  #define __DIVPROC_H__
3
4  #include "set.h"
5  #include <mpi.h>
6
7  class DivProc {
8  private: // Internal data
9    int rank;
10   Set m_SET;
11 private: // Internal behaviour
12   void Send(int dst, int value) {
13     MPI_Send(&value,1,MPI_INT,dst,100,MPI_COMM_WORLD);
14     printf("Proc#%i: sent          = %i\n",rank,value);
15   };
16   void Recv(int dst, int& value) {
17     MPI_Status status;
18     MPI_Recv(&value,1,MPI_INT,dst,100,MPI_COMM_WORLD,&status);
19     printf("Proc#%i: received      = %i\n",rank,value);
20   };
21   void Small();
22   void Large();
23 public: // Constructor/destructor
24   DivProc() {};
25   ~DivProc() {};
26 public: // Public methods
27   void Init(int argc, char *argv[]) {
28     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
29     m_SET.InitCmdLine(argc, argv);
30   };
31   void Run(void);
32 };
33
34 #endif
```

divproc.cpp

```
1  #include "divproc.h"
2  #include <stdio.h>
3
```

```
4  #define SMALL_PROC 0
5  #define LARGE_PROC 1
6
7  void DivProc::Run(void) {
8    printf("Proc#%i (before): ",rank); m_SET.Print(); printf("\n");
9    if (rank==SMALL_PROC) Small(); else
10   if (rank==LARGE_PROC) Large();
11   printf("Proc#%i (after) : ",rank); m_SET.Print(); printf("\n");
12 }
13 void DivProc::Small() {
14   int mx, x;
15   mx=m_SET.max(); Send(LARGE_PROC,mx); m_SET-mx;
16   Recv(LARGE_PROC,x); m_SET+x; mx=m_SET.max();
17   while (mx>x) {
18     Send(LARGE_PROC,mx); m_SET-mx;
19     Recv(LARGE_PROC,x); m_SET+x; mx=m_SET.max();
20   }
21 }
22 void DivProc::Large() {
23   int mn, y;
24   Recv(SMALL_PROC,y); m_SET+y; mn=m_SET.min();
25   Send(SMALL_PROC,mn); m_SET-mn; mn=m_SET.min();
26   while (mn < y) {
27     Recv(SMALL_PROC,y); m_SET+y; mn=m_SET.min();
28     Send(SMALL_PROC,mn); m_SET-mn; mn=m_SET.min();
29   }
30 }
```

main.cpp

```
1  #include <mpi.h>
2  #include "divproc.h"
3
4  int main(int argc, char *argv[]) {
5    DivProc proc;
6    MPI_Init(&argc, &argv);
7    proc.Init(argc, argv);
8    proc.Run();
9    MPI_Finalize();
10   return 0;
11 }
```

REFERENCES

[1] Lafortune, S.—Wang, Y.—Reveliotis, S.: Eliminating Concurrency Bugs in Multithreaded Software: An Approach Based on Control of Petri Nets. In: Colom, J.-M., Desel, J. (Eds.): Application and Theory of Petri Nets and Concurrency (PETRI NETS 2013). Lecture Notes in Computer Science, Vol. 7927, 2013, pp. 21–28, doi: 10.1007/978-3-642-38697-8_2.

[2] Wang, Y.—Kelly, T.—Kudlur, M.—Lafortune, S.—Mahlke, S.: Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08), Berkeley, CA, USA, USENIX Association, 2008, pp. 281–294.

[3] Böhm, S.—Běhálek, M.—Meca, O.—Šurkovský, M.: Kaira: Development Environment for MPI Applications. In: Ciardo, G., Kindler, E. (Eds.): Application and Theory of Petri Nets and Concurrency (PETRI NETS 2014). Lecture Notes in Computer Science, Vol. 8489, 2014, pp. 385–394.

[4] Pelayo, F. L.—Cuartero, F.—Valero, V.—Macia, H.—Pelayo, M. L.: Applying Timed-Arc Petri Nets to Improve the Performance of the MPEG-2 Encoding Algorithm. Proceedings of the 10th International Multimedia Modelling Conference (MMM'04), Washington, DC, USA, 2004, p. 49, doi: 10.1109/MULMM.2004.1264966.

[5] Kristensen, L. M.: An Approach for the Engineering of Protocol Software from Coloured Petri Net Models: A Case Study of the IETF WebSocket Protocol. International Workshop on Petri Nets and Software Engineering (PNSE 2014). CEUR Workshop Proceedings, Vol. 1160, 2014, pp. 13–14.

[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. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 8100, 2013, pp. 71–88.

[7] Westergaard, M.: Verifying Parallel Algorithms and Programs Using Coloured Petri Nets. In: Jensen, K., van der Aalst, W. M., Ajmone Marsan, M., Franceschinis, G., Kleijn, J., Kristensen, L. M. (Eds.): Transactions on Petri Nets and Other Models of Concurrency VI. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 7400, 2012, pp. 146–168, doi: 10.1007/978-3-642-35179-2_7.

[8] Voron, J. B.—Kordon, F.: Transforming Sources to Petri Nets: A Way to Analyze Execution of Parallel Programs. Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops (Simutools '08), ICST, Brussels, Belgium, 2008, Art. No. 13, doi: 10.4108/ICST.SIMUTOOLS2008.3055.

[9] Dijkstra, E. W.: A Correctness Proof for Communicating Processes: A Small Exercise. Selected Writings on Computing: A Personal Perspective. Texts and Monographs in Computer Science, Springer, New York, NY, 1982, pp. 259–263.

[10] Karpov, Yu. G.—Borshchev, A. V.: On the Correctness of Parallel Algorithms. Programming and Computer Software, Vol. 22, 1996, pp. 164–171.

[11] Anisimov, N. A.—Golenkov, E. A.—Kharitonov, D. I.: Compositional Petri Net Approach to the Development of Concurrent and Distributed Systems. Programming and Computer Software, Vol. 27, 2001, No. 6, pp. 309–319.

[12] Kharitonov, D.—Tarasov, G.: Modeling Function Calls in Program Control Flow in Terms of Petri Nets. ACSIJ Advances in Computer Science: An International Journal, Vol. 3, 2014, No. 6, pp. 82–91.

[13] Apt, K. R.—Francez, N.—de Roever, W. P.: A Proof System for Communicating Sequential Processes. ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 2, 1980, No. 3, pp. 359–385.

[14] Peterson, J. L.: Petri Net Theory and the Modeling of Systems. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[15] Reisig, W.: Petri Nets: An Introduction. Springer-Verlag New York, Inc., New York, NY, USA, 1985, doi: 10.1007/978-3-642-69968-9.

[16] Jensen, K.: Coloured Petri Nets. In: Brauer, W., Reisig, W., Rozenberg, G. (Eds.): Petri Nets: Central Models and Their Properties: Advances in Petri Nets 1986, Part I Proceedings of an Advanced Course, Bad Honnef, September 8–19, 1986. Lecture Notes in Computer Science, Vol. 254, 1987, pp. 248–299, doi: 10.1007/978-3-540-47919-2_10.

[17] Lakos, C.: From Coloured Petri Nets to Object Petri Nets. In: De Michelis, G., Diaz, M. (Eds.): Application and Theory of Petri Nets 1995 (ICATPN 1995). Lecture Notes in Computer Science, Vol. 935, pp. 278–297, doi: 10.1007/3-540-60029-9_45.

[18] Maier, C.—Moldt, D.: Object Coloured Petri Nets – A Formal Technique for Object Oriented Modelling. In: Agha, G. A., De Cindio, F., Rozenberg, G. (Eds.): Concurrent Object-Oriented Programming and Petri Nets. Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 2001, 2001, pp. 406–427.

[19] Lilius, J.: $OB(PN)^2$: An Object Based Petri Net Programming Notation. Technical Report, Turku Centre for Computer Science, 1999.

Dmitriy Kharitonov is a senior research fellow at Institute of Automation and Control Processes (IACP), Russian Academy of Sciences, Vladivostok, Russian Federation. He received his M.Sc. degree in applied mathematics from Moscow Institute of Physics and Technologies in 1996 and his Ph.D. degree in software of computational networks and systems from IACP in 2001. He is interested in program verification, program modeling, parallel programming languages and translators.

George Tarasov is a research fellow at IACP. He received his M.Sc. degree in automation control systems from Far-Eastern Federal University, Vladivostok, Russia in 1999. Since 2001 he teaches courses at Far-Eastern Federal University on parallel programming and theory of computing processes and structures. He is interested in parallel programming, program verification and performance analysis.

Evgeniy Golenkov is a senior research fellow at IACP. He has worked in the field of information technologies since 1971 when he graduated from the Moscow Institute of Physics and Technology (MIPT). He received his Ph.D. degree from the Institute of Cybernetics (Ukraine) in 1975. From 1976 up to 1990 he taught at MIPT. Since 1990 till now he has taught at Far-Eastern Federal University. He is interested in computer network software and supercomputing technologies. He took part in 15 science and educational projects, and published more than 80 articles.