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

Dmitriy Kharitonov and George Tarasov
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 \( A = \{a_1, a_2, \ldots, a_k\} \) is a set. Multiset on \( A \) is defined as a function \( \mu: A \to \{0, 1, 2, \ldots\} \), that associates with each element of set \( A \) some non-negative integer number. Multisets are conveniently written as a formal sum.
\[ n_1a_1 + n_2a_2 + \cdots + n_ka_k \] 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 \( 1 \leq i \leq k \). We say \( \mu_1 \leq \mu_2 \), if \( n_i \leq m_i \) for each \( 1 \leq i \leq k \), and \( \mu_1 < \mu_2 \), if \( \mu_1 \leq \mu_2 \) and \( \mu_1 \neq \mu_2 \). If \( n_i = 0 \) for all \( i \), than this multiset is denoted as \( \emptyset \). 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 \( M(A) \).

**Definition 1:** Petri net is a tuple \( \Sigma = (S, T, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot) \), where
1. \( S \) is a finite set of places;
2. \( T \) is a finite set of transitions, with \( S \cap T = \emptyset \);
3. \( \cdot : T \rightarrow M(S) \) is an incoming incidence function;
4. \( \cdot : T \rightarrow M(S) \) is an outgoing incidence function.

Multisets \( *t \) and \( t^* \) are referred to as incoming and outgoing 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 \( E = (\Sigma, \Gamma, M_0) \), where
1. \( \Sigma = (S, T, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot) \) is a object structure (Petri net);
2. \( \Gamma = \{a_1, a_2, \ldots, a_\alpha\} \) is a set of access points (AP), each having form \( \alpha = (id_\alpha, \Delta_\alpha, \sigma_\alpha) \), where
   - \( id_\alpha \) is a name of access point,
   - \( \Delta_\alpha \) is an alphabet and
   - \( \sigma_\alpha : T \rightarrow M(\Delta_\alpha) \) is a transition labeling function;
3. \( M_0 \in M(S) \) 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 \( \alpha, \beta, \gamma \). The following graphical notations are used for objects. PN-object is a Petri net, 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 \( \alpha \) will be defined as \( \alpha(t) \equiv \sigma_\alpha(t) \).

\( id_\alpha \) 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:
\[ \forall t \in T, \forall \alpha, \alpha_2 \in \Gamma : \alpha_1(t) \neq \emptyset \land \alpha_2(t) \neq \emptyset \Rightarrow \alpha_1 = \alpha_2. \]

**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 \( E_1 = (\Sigma_1, \Gamma_1, M_{0_1}) \) and \( E_2 = (\Sigma_2, \Gamma_2, M_{0_2}) \), where \( \Sigma_1 = \langle \iota_1, \iota_1, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot \rangle \) and \( \Sigma_2 = \langle \iota_2, \iota_2, \cdot, \cdot, \cdot, \cdot, \cdot \rangle \). Operation of formal union of objects \( E_1 \) and \( E_2 \) creates new object \( E = E_1 \oplus E_2 = (\Sigma, \Gamma, M_0) \), where \( \Sigma = (S, T, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot) \) and \( S = S_1 \cup S_2 \), \( T = T_1 \cup T_2 \), \( M_0 = M_{0_1} + M_{0_2} \), \( \cdot = \cdot \cup \cdot \cup \cdot \), \( \cdot = \cdot \cup \cdot \cup \cdot \), \( \Gamma = \Gamma_1 \cup \Gamma_2 \).

**Definition 4:** Consider PN-object \( E_1 = (\Sigma_1, \Gamma_1, M_{0_1}) \) that have two access points \( \alpha = (id_\alpha, \Delta_\alpha, \sigma_\alpha) \in \Gamma_1 \) and \( \beta = (id_\beta, \Delta_\beta, \sigma_\beta) \in \Gamma_1 \). Operation of union of access points \( \alpha \) and \( \beta \) of \( E_1 \) creates new PN-object \( E = (E_1)_{\alpha = \beta} = (\Sigma, \Gamma, M_0) \) where \( \Sigma = \Sigma_1 \cup \{\alpha, \beta\} \cup \{\gamma\}, M_0 = M_{0_1} \cup \gamma \), \( \gamma = (id_\gamma, \Delta_\gamma, \sigma_\gamma) \), \( \gamma = (id_\gamma, \Delta_\gamma, \sigma_\gamma) \), \( \sigma_\gamma(t) = \sigma_\alpha(t) \cup \sigma_\beta(t) \). The value for \( id_\gamma \) can be chosen freely. Some simple algorithms are acceptable for further operations:
- \( id_\gamma = \gamma \),
- \( id_\gamma = id_\alpha \lor id_\beta \lor id_\gamma = id_\alpha \land id_\gamma = id_\beta \).

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 \( E_1 = (\Sigma_1, \Gamma_1, M_{0_1}) \), where \( \Sigma_1 = \langle \iota_1, \iota_1, \cdot, \cdot, \cdot, \cdot \rangle \) and \( \alpha = (id_\alpha, \Delta_\alpha, \sigma_\alpha) \in \Gamma_1 \). Operation of restriction of object \( E_1 \) by access point \( \alpha \) creates new object \( E = (\Sigma, \Gamma, M_0) = \sigma_\alpha(E_1) \), where \( \Sigma = (S, T, \cdot, \cdot, \cdot, \cdot) \) that, \( S = S_1 \cup T_1 \setminus \{t \in T \mid \sigma_\alpha(t) > 0\} \), \( \forall t \in T | \cdot t = \cdot t_1, \cdot t = \cdot t_2, M_0 = M_{0_1} \setminus \alpha \).
Restriction of object by access point deletes each transition having label from $\Delta(\alpha)$ with all adjacent arcs.

**Definition 6:** Consider PN-object $E_1 = (\Sigma_1, \Gamma_1, M_{01})$ and its access points $\alpha, \beta \in \Gamma_1$, where $\alpha = (\text{id}_{\alpha}, \Delta_{\alpha}, \sigma_{\alpha})$, $\beta = (\text{id}_{\beta}, \Delta_{\beta}, \sigma_{\beta})$ and $\Sigma_1 = (S_1, T_1, \cdot \{1\}, \cdot \{1\})$. Operation of simple composition of object $E_1$ by access points $\alpha$ and $\beta$ forms new object $E = (\Sigma, \Gamma, M_0)$, where $\Gamma = \Gamma_1, M_0 = M_{01}$ and $\Sigma = (S, T, \cdot \{0\}, \cdot \{0\})$:

1. $S = S_1$.
2. $T = T_1 \cup T_{\text{syn}}$ where $T_{\text{syn}} = \{\mu_1 + \mu_2 | \mu_1, \mu_2 \in M(T_1)\}$, and $\sigma_{\alpha}(\mu_1) = \sigma_{\beta}(\mu_2) > \emptyset$, 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. $\cdot \{0\} = \cdot \{0\} \cup \{(\mu_1 + \mu_2, (\mu_1)\uplus(\mu_2)\}| \mu_1 + \mu_2 \in T, \mu_1, \mu_2 \in M(T_1)\}$.
4. $(\cdot \{0\} = \cdot \{0\} \cup \{(\mu_1 + \mu_2, (\mu_1)\uplus(\mu_2)\}| \mu_1 + \mu_2 \in T, \mu_1, \mu_2 \in M(T_1)\}$.
5. $\forall t \in T_{\text{syn}} \forall \xi \in \Gamma : \xi(t) = 0$.

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

\[
E = [E_1]_{\beta}. \quad E_1 \uplus \alpha \beta E_2 \equiv [E_1 \uplus E_2]_{\beta}
\]

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

\[
\cdot (\mu_1 + \mu_2) = \cdot (\mu_1) \uplus \cdot (\mu_2) \quad (\mu_1 + \mu_2)^* = (\mu_1)^* \uplus (\mu_2)^*.
\]

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

\[
E = a(\eta)_{\beta} = \partial_a[(E_1)_{\beta}^\beta].
\]

For two objects $E_1 = (\Sigma_1, \Gamma_1, M_{01})$ and $E_2 = (\Sigma_2, \Gamma_2, M_{02})$ with access points $\alpha \in \Gamma_1$ and $\beta \in \Gamma_2$ accordingly, operation of directional composition (binary form) is denoted as

\[
E = E_1 \rightarrow_\alpha E_2 = \partial_a(E_1 a[]_{\beta} E_2).
\]

This definition implies that a unary directional composition performs two operations on an object simple composition by AP $\alpha$ and $\beta$ and restriction by AP $\alpha$. 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:

\[
E_1 \uplus E_2 = E_2 \uplus E_1,
\]

\[
E_1 \uplus (E_2 \uplus E_3) = (E_2 \uplus E_3) \uplus E_1.
\]

**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:

\[
(\alpha + \beta)_{\gamma} = (\beta + \alpha)_{\gamma},
\]

\[
((\alpha + \beta)_{\gamma} + \gamma)_{\zeta} = ((\beta + \alpha)_{\gamma} + \gamma)_{\zeta}.
\]

**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:

\[
\partial_a((\phi(E)) = \partial_b((\phi(E)).
\]

**Proof:** Follows from definition 5. As far as subsets of labeled transitions after restrictions by $\alpha$ and $\beta$ do not intersect, then in both cases we get the same set $T$, and hence the equality is fair.

4. For two objects $E_1 = (\Sigma_1, \Gamma_1, M_{01})$ and $E_2 = (\Sigma_2, \Gamma_2, M_{02})$ and their access points $\zeta \in \Gamma_1$ and $\xi \in \Gamma_2$, next equality holds:

\[
E = E_1 \mid \zeta \partial_a(E_2) = \partial_a(E_1 \mid \zeta E_2).
\]

**Proof:** Follows from definitions 3 and 5.

5. Associativity of directional objects composition:

\[
\alpha \beta \gamma \alpha \beta \gamma
\]

\[
E_1 \rightarrow_{\alpha}(E_2 \rightarrow_{\gamma} E_3) = E_2 \rightarrow_{\gamma}(E_1 \rightarrow_{\alpha} E_3).
\]

**Proof:** Follows from definitions 3 and 5.
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 \( E_{1, \beta} \), where the set of access points consists from \( \Pi = \{ \pi_1, \ldots, \pi_n \} \) and the access point \( \alpha \), the next notation will be used:

\[
E = \pi_n \xrightarrow{\alpha} \ldots \pi_1 \xrightarrow{\alpha} E_{1, \beta} = \pi_1. 
\]

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 \( \alpha \) and \( \beta \) access points, and restriction by \( \alpha \) 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;

Fig. 1. Example of directional composition: (a) original representation, (b) result of objects formal union, (c) result of simple composition, (d) result of restriction.
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 \( E = \langle \Sigma, \Gamma, M_0 \rangle \), \( \Sigma = \langle S, T, \ast \rangle \), \( \Gamma = \langle \ast \rangle \) and alphabet \( \Delta_f = \{ \text{begin}_f, \text{end}_f \} \). PN-object has access point \( \alpha \in \Gamma \), in form \( \alpha = (f, \Delta_f, \sigma_f) \).

Then three elements \( (t_1, s, t_2) \) 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. \( t_1^* = s = t_2, s^* = t_1, s = t_2 \).

will be called model of the call of function \( f \) in the context of access point \( \alpha \), if
\[
\sigma_f(t_1) = \text{begin}_f, \sigma_f(t_2) = \text{end}_f.
\]

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 \( \text{begin}_f \)), this transition is called call transition;
- place models a state of function completion waiting;
- second transition models control flow return from the function (label \( \text{end}_f \)), 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 \( f \) call is described by a separate alphabet \( \Delta_f \) and a separate access point \( \sigma_f \). 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 \( E_f = \langle \Sigma_f, \Gamma_f, M_{0_f} \rangle \), where \( \Sigma_f = \langle S_f, T_f, \ast \rangle \) – object structure, describing control flow of the function body, and there is in \( \Gamma_f \) two disjoint subsets of access points
\[
\mathcal{P}_f = \{ \text{in} \} \text{ is a subset of incoming access points, consisting of one access point } \text{in} = (f, \Delta_f, \sigma_f), \Delta_f = \{ \text{begin}_f, \text{end}_f \}. \text{ OUT}_f = \{ \text{out}_1, ..., \text{out}_n \} \text{ is a subset of outgoing access points, satisfying the definition 8. If for the object structure } \Sigma_f \text{ and for access points } \text{in}, \text{out}_1, ..., \text{out}_n \text{ next statements are fair:}
\]
1. \( \exists! t_b \in T \colon \exists t_b = \emptyset \wedge \text{in}(t_b) = \text{begin}_f \),
2. \( \forall t \in T_f : t \neq t_b \wedge t^* = \emptyset \wedge \text{in}(t) = \text{end}_f \), where \( T_f \subset T \) and \( T_f \neq \emptyset \);
3. \( \forall \text{out}_i \in \text{OUT}_f : \text{out}_i(t_k) = \text{begin}_f \Rightarrow \text{out}_i(t_k) = \text{begin}_f \Rightarrow \text{out}_i(t_k) = \text{end}_f \);

then this PN-object \( E_f \) will be referred as function \( f \) model.

\[
E_f[\alpha, \beta] = \begin{cases} \text{begin}_f, & \alpha = 1, \beta = \emptyset, \\ \text{end}_f, & \alpha = \emptyset, \beta = 1, \\ \text{call}, & \alpha = \emptyset, \beta = \emptyset. \end{cases}
\]

Let’s use additional notations: \( \text{In}(E_f) = \mathcal{P}_f \) - function \( \text{In} \) returns a set of incoming access points of PN-object, \( \text{Out}(E_f) = \text{OUT}_f \) - function \( \text{Out} \) returns, accordingly, a set of outgoing access points of PN-object.

Examples of function models and function call models are shown on a Fig. 3. Objects \( E_f \), \( E_{f_1} \) and \( E_{f_2} \) represent models of three different functions \( f, f_1 \) and \( f_2 \) accordingly. Each object has one incoming access point \( \text{in}_1 \), from which control flow is transferred to function body, modeled by object. Object \( E_f \) has two outgoing access points describing calls of functions \( f_1 \) and \( f_2 \). 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 $s_3$). Then, in object $E_f$ there can be fired transition $t_2$, labeled $\text{begin}_{f_1}$. In accordance with the rules of directional composition operation, objects $E_f$ and $E_{f_1}$ are combined together, and from transitions $t_3 (E_f)$ and $t_{b_2} (E_{f_1})$ a new transition is formed, that removes token from $s_2$ and puts one token to the places $s_2 (E_f)$ and $s_8 (E_{f_1})$. Token in place $s_2$ means waiting of control flow return from function $E_{f_1}$ (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 $t_{o_4}$ in object $E_{f_1}$.

Let’s generalize the notion of functions 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 $E = (\Sigma, \Gamma, M_0)$, that have defined functions $\text{In}$ and $\text{Out}$. Without loss of generality, it is assumed that $|\text{In}(E)| = 1, |\text{Out}(E)| \geq 0$.

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 outgoing 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 $\text{SELF}_F \subset \Gamma_F$ that includes all output access points of the module, having alphabets included in the alphabet of input access point for this module:

$$\text{SELF}_F = \{\text{out}_i | \text{out}_i \in \text{Out}(F) \land \Delta(\text{out}_i) \subseteq \Delta(\text{In}(F))\}.$$  

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 $E = (\Sigma, \Gamma, M_0)$ such, that:

$$E = E_1 \uplus E_2 = \text{SELF}_{(E_1 \uplus E_2) \setminus \text{in}_1 + \text{in}_2} = \text{SELF}_{(E_1 \uplus E_2) \setminus \text{in}_1 + \text{in}_2}$$

where $\text{In}(E) = (\text{in})$ incoming access point of new module, $\text{Out}(E) = (\text{Out}(E_1) \cup \text{Out}(E_2)) \setminus \text{SELF}$ - set of new module outcoming access points, $\text{in}_1$ and $\text{in}_2$ - 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 $\text{in}_1$, $\text{in}_2$ and subsets of outcoming access points $\{f_1, f_2\}$ and $\{f_3, f_4, g\}$ accordingly. It is assume 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 $\text{in}_1$ and $\text{in}_2$ is performed, subset of internal access points SELF is highlighted, consisting of $\{g\}$, then directional composition by access points from sets $\text{SELF} = \{g\}$ and $\text{In} = \{\text{in}\}$ 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. 3. Examples of function models and function call models.

Fig. 4. Examples 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 \((f_1, f_2, f_3)\). 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 \((\text{program entrance point})\). 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 \(E = (\Sigma, \Gamma, M_0)\), where \(\Sigma = (S, T, \cdot, \cdot, \cdot)\) and \(\Gamma = \{\alpha\} : \alpha = (\text{out}, \Delta, \sigma)\) and \(\Delta = \{\text{begin, end}\}\). We will call \(E\) the model of loader, if:

- \(\exists ! s_b, s_e \in S : s_b \neq s_e, s_b^* = 0, s_e^* = 0, \)
- \(\forall s \in S : M_0(s) = \begin{cases} 1, \text{if } s = s_b, \\ 0, \text{otherwise,} \end{cases} \)
- \(\exists ! (t_1, s, t_2), t_1, t_2, s, t \in S : s \neq s_b \neq s_e \land t_1^* = s = t_2^* \) and \(\text{out}(t_1) = \text{begin}\) and \(\text{out}(t_2) = \text{end} \).

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 \(\text{out}\) 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:
- \(\text{out}(E_1) = \{\alpha\} , \text{in}(E) = \{\beta\} , \alpha = (\text{out}, \Delta, \sigma_\alpha), \beta = (\text{in}, \Delta, \sigma_\beta)\),
- \(\Delta_\alpha = \Delta_\beta = \{\text{begin, end}\}\).

Then object \(E = \partial_\beta(E_1 \rightarrow E_2)\) 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.
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 or 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
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 a waiting 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


Dmitry 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.