IMPERATIVE PROGRAMS BEHAVIOR SIMULATION IN TERMS OF COMPOSITIONAL PETRI NETS

Leontyev Denis Vasilevich, Kharitonov Dmitry Ivanovich and Tarasov Georgiy Vitalievich

Institute of Automation and Control Processes FEB RAS Vladivostok, Russia

ABSTRACT

The article considers a generation mechanism of compositional models simulating imperative programs behavior in terms of Petri nets. The mechanism of program models generation consists of two main stages. At the first stage, the structure of the program is prepared using such program elements like: libraries, functions and links between functions. At the second stage, the content of function bodies is generated on the base of template constructions. In the article some semantic constructions template examples of imperative programming language with their descriptions are given, and a generation example of a program model in terms of Petri nets is demonstrated.

KEYWORDS

Program modeling; control flow; Petri nets; composition operations

1. INTRODUCTION

Petri nets are a well known formal theory used to describe various distributed systems. In particular, Petri nets are widely used for program modeling. A reach ability tree of a Petri net is its state space representation and one of the main tools for Petri nets analyzing [1, 2]. It is a set of states reachable from the initial marking, interconnected either by triggered transitions or by triggered steps, composed by a multiset of simultaneously fired transitions. The algorithm for a reach ability tree construction in every iteration performs a search for all excited transitions, triggers chosen transitions, and adds newly constructed Petri net markings to the state space. In case when we are dealing with relatively small Petri nets, this procedure is quite simple. However, increasing the size of the net causes algorithmic problems associated with the net placement and the state space representation in memory, then with the effectiveness of reach ability tree duplicate states elimination and with the overall time needed to build the reach ability tree [3, 4]. Given into consideration than the number of places and transitions in Petri net modeling programs is several times the number of program source code lines, this means that for a program consisted of one megabyte of source code, the model in terms of Petri nets can

The work was funded by presidium of Russian Academy of sciences Fundamental research program grant № 0262 2015 0139. Have roughly about 106 elements, and for a programs of real complexity it can be even ten more times larger. Representation in the form of incidence matrix of such Petri nets in computer memory is resource intensive and inefficient. Taking into account that the “mobility” of tokens in program modeling Petri nets is related to the computational processes in the program, the standard algorithm of reach ability tree construction can be optimized both to reduce the required amount of RAM and to speed up the building process in multiple times. However in practice, building such large program models is still not a routine process [5]. For this matter an automated mechanism for generating Petri nets simulating programs of the required size with predetermined characteristics is required that is to provide experimental base for development of algorithms and data formats for large scale Petri nets processing. In this paper, the authors propose such generation mechanism based on templates [6]. Earlier, the authors developed a mechanism that allows building so called simple Petri nets, and now this mechanism was adapted to generate compositional models, i.e. models built from the composition of Petri nets.

2. ALGORITHM for BUILDING COMPOSITIONAL MODELS of PROGRAMS in TERMS Of PETRI NETS

In the process of programs development, such structural elements as libraries and functions are widely used. It seems natural that in a building of models simulating real programs, such structural elements must also be involved. One way to do this is building models compositionally. This allows to take into account the program structure, and ensures the reuse of structural elements and, as one of consequences, reduces necessary RAM amount. An indirect advantage of this approach is the ability to develop a parallel version of this building algorithm without many efforts.

Later on, we need to distinguish between two kinds of Petri nets. A Petri net of the first kind or simply a Petri net is a tuple consisting of: a set of places, a set of transitions, input and output functions of transitions incidence (multi-set of places necessary for transitions firing). For the purposes of this article, Petri nets of the first kind are generated using template constructions. Template constructions are merging with each other by a places composition operation.

Petri nets of the second kind composed of Petri nets of the first kind and Petri nets of the second kind by means of a transition composition operation.

The merging of two Petri nets requires the assignment of access points. There are two types of them: access points by places and access points by transitions. Within the scope of this article, access points by places are simply called access points, and access points by transitions are called compositional access points.

Fig. 1 shows an overview of the algorithm for building compositional models of programs in terms of Petri nets. A program model building process can be described as follows:

1. In the first stage, a configuration of the generated program model is read. The configuration describes template constructions (or just templates) and their parameters (probability of the template selection, the minimum number of uses), the configuration of libraries, and the number of elements in the model resulted. Each library configuration consists of: the number of functions, the minimum and the maximum size of library functions; the number of library internal functions and the number of recursive calls within the library.

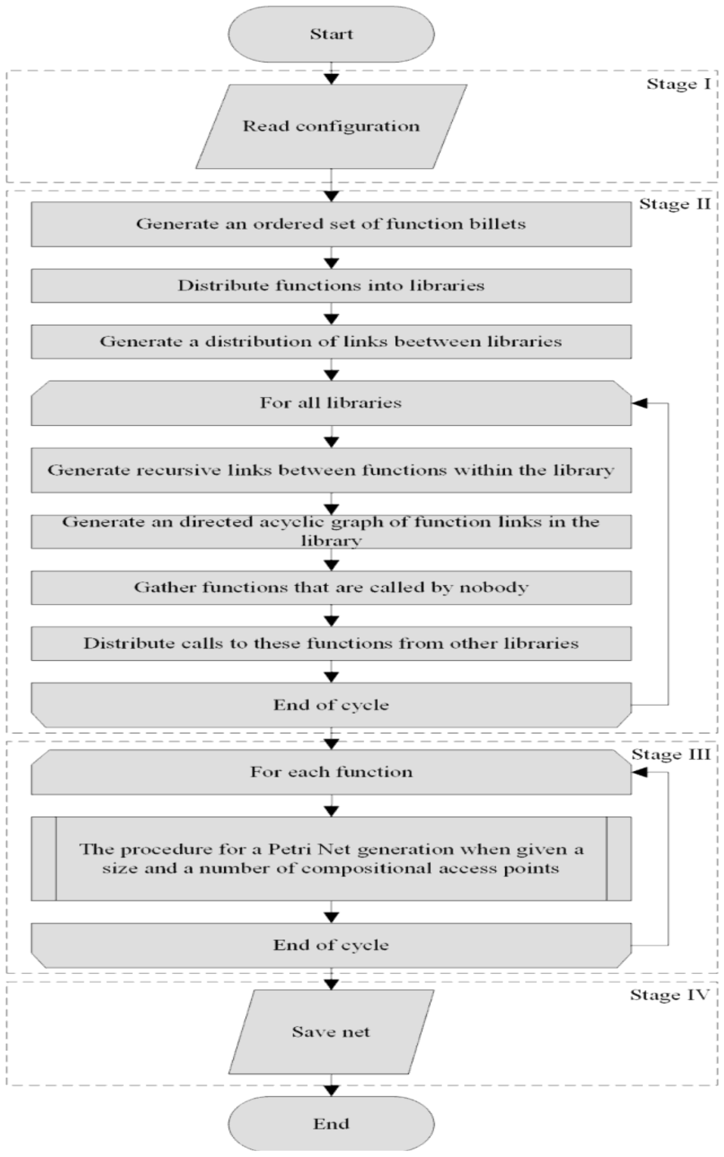

Start
Stage I
Read configuration
Stage II
Generate an ordered set of function billets
Distribute functions into libraries
Generate a distribution of links beetween libraries
For all libraries
Generate recursive links between functions within the library
Generate an directed acyclic graph of function links in the library
Gather functions that are called by nobody
Distribute calls to these functions from other libraries
End of cycle
Stage III
For each function
The procedure for a Petri Net generation when given a size and a number of compositional access points
End of cycle
Stage IV
Save net
End

Fig. 1 Algorithm of program composite models generation in terms of Petri nets.

2. At the second stage, the structure of the program model is built, i.e. a set function billets is created, then divided into libraries, and provided with links between functions and libraries. The details of the procedure are described below in the text.
3. The third stage is the building of each function body as a Petri net of the first kind. The building procedure is also described below in the text.
4. And at the final fourth stage, the resulting net is written to a file.

Let us consider the second and the third stages in details:

In the first step, an ordered set of function billets is organized in an array and an input access point is added to each function for the subsequent construction of the function body. This is necessary for the addressing functions by their index in array and is convenient for grouping functions into libraries by indexes range. This approach also allows making partial order of libraries and functions, that enables calls from a higher-level library to the functions of lower-level libraries only, but not the other way around.

The next step simulates separation of functions into libraries, which occurs when building real programs using multiple libraries. So some libraries are basic, other libraries interact with basic ones, and eventually the program uses libraries providing high level functions. The use of a library functions from another is modeled by the links between libraries, which are directed from the higher level libraries to the lower level libraries. This step selects from earlier read range the size of each function (to satisfy the number of elements in the resulting model), and specifies the number of library internal functions, i.e. number of those functions that do not call others.

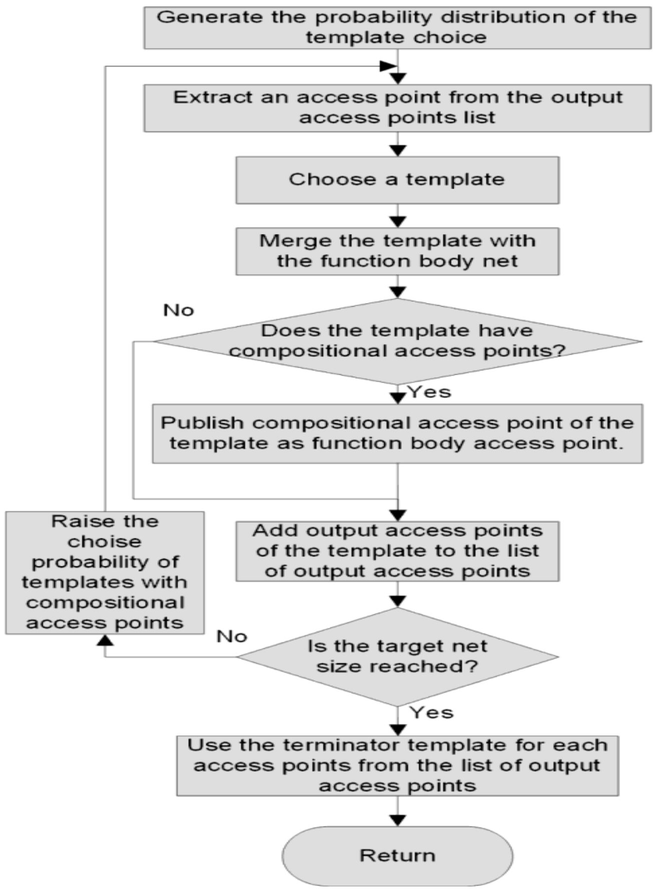
Inside the libraries interactions between functions are also possible, and some function calls can be recursive, either direct (the function calls itself) or indirect (the function calls other functions, that call functions … and finally the first function is called). Indirect and direct recursions are possible only within one library and must be restricted to resemble human code.

Building links between functions in libraries should guarantee that all functions are called and there are no “hanging free” functions. This is be ensured by adding all recursive calls separately from non recursive, that are produced by procedure of function connections directed acyclic graph construction. The building of the graph begins from the low level functions in the direction to the high level functions, adding necessary output access points to the functions list of access points.

After acyclic graph is done, it may be possible that some library functions are called from nobody. These functions must be called from higher-level libraries that is performed by a simple distribution procedure.

This process is repeated until all libraries are covered.

In the third stage, the procedure for constructing the function body is performed. The function body construction algorithm is shown in the Fig. 2.The Function body is represented by the algorithm as a Petri net of the first kind. In general, the building process performs consequently merging of the semantic construction templates by the places composition operation. A semantic construction template can have one input and some output access points. In addition, each access point has a number of places that are to be merged with another template places. The number of places is called the power of access point. Output access points can be merged only with input access points and their powers must be equal.



Generate the probability distribution of the template choice
Extract an access point from the output access points list
Choose a template
Merge the template with the function body net
Does the template have compositional access points?
No
Yes
Publish compositional access point of the template as function body access point.
Add output access points of the template to the list of output access points
Raise the choise probability of templates with compositional access points
Is the target net size reached?
No
Yes
Use the terminator template for each access points from the list of output access points
Return

Fig. 2. The Petri net generation procedure when given a size and a number of compositional access points.

At the beginning of the stage, the probability distribution of templates selection is built. This is necessary for the algorithm to have a predictable influence on behavioral properties of the resulting net. For example, to make a model resembling tree like programs, one can specify a higher probability of branching semantic construction template in the settings file.

Iterative part of the third stage continues while there are access points in the list of unsatisfied output access points. Each iteration extracts one access point from the list. Then the choice of the semantic construction template is made based on the type and power of chosen access point. Templates can have access points of two types: by places and by transitions. Templates having access point by transitions are called compositional. It is worth noting that templates can be merged only by access points of the same type. Templates examples will be discussed later.

The next step is to merge the semantic construction template with the body net. All the output access points of the semantic construction template are added to the list of the body net output access points. If the selected template is compositional, it is also necessary to “publish” the output compositional access point of the template as the output access point of the function body. Compositional access points inside the body are used to represent the calls to external functions, which must also be satisfied.

The Final step of the iteration is to check the target net size and update the probability distribution of templates. While the net target size is not obtained, it increases the probability of selecting a compositional template. After all compositional access points of the function body are satisfied these probabilities are zeroed. Dynamic updates of a probability distribution are necessary in order to ensure that when the target net size is reached, all compositional access points are satisfied.

3. SEMANTIC CONSTRUCTION TEMPLATES IN TERMS OF PETRI NETS

Program models are constructed using templates of semantic constructions of imperative programming languages. Fig. 3 shows eleven templates that are sufficient for generation of a model that simulates the behavior of a consistent imperative program. The templates do not depend on the algorithm, i.e. the templates can be easily changed in the configuration file without any changes in the source code.

The following designations are used in template representations. Petri nets are drawn usual graphical notation in the form of a bipartite directed graph, where places are represented by circles and transitions by rectangles. Places and transitions are connected by arcs. Petri net describing the structure of the template is placed in a rectangle. At the boundaries of the rectangle are drawn symbolic images of access points by places in the form of a circle and by transitions in the form of a triangle. Input access points by places are drawn on the top of the rectangle, and the output ones   on the bottom. All the input compositional access points are drawn by a triangle directing one vertex inside the rectangle, and the output - directing one vertex outside.

The template "A" called the loader is designed to imitate the begin and the end of a sequential process. This is the only start template in the described set that has no input interface. The initial place of the loader template has a token and represents the starting point of the program model where the program starts its work. The template has one output compositional access point that represents the call to the main function. This access point allows the template to merge only with the template "C".

Template "B" imitates a function call in the imperative program. This template has a single access point in input interface and two access points in the output interface. The first access point of the output interface is designed to represent how the function is called from other functions. This compositional access point consists of a transition pair so that through this access point, it is

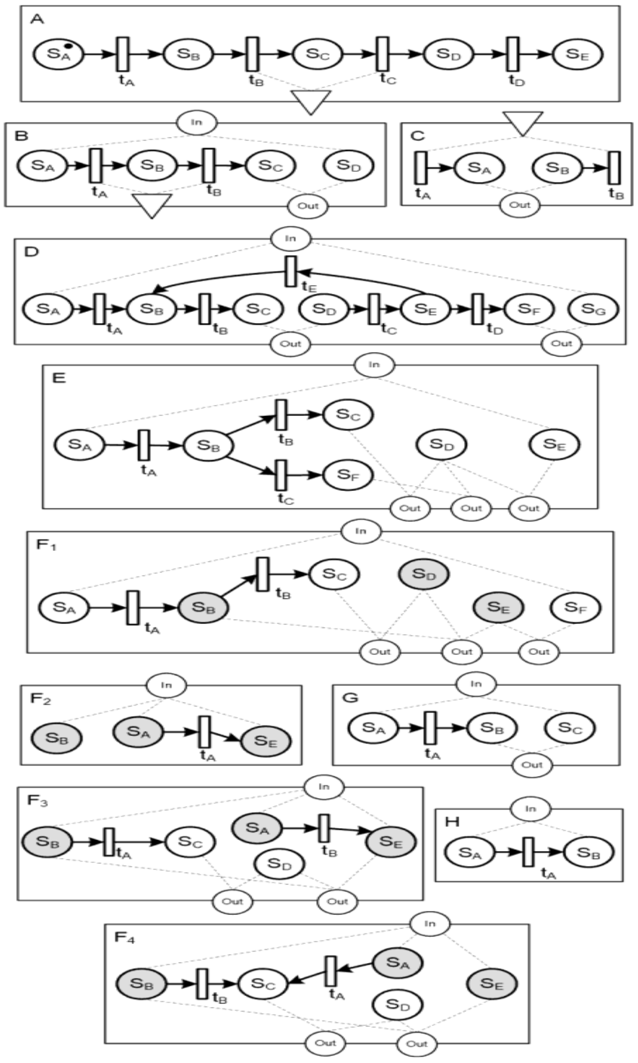


Fig. 3. Semantic constructions templates of imperative programming language in terms of Petri nets.

Possible to merge only with the template "C". The second access point is designed to continue the program after the function call and consists of two places.

Template "C" represents the function body. This template has one input and one output access point. The entry point is a compositional access point. The output access point is designed for generating a body.

The template "D" simulates cycles. The first access point of the template output interface is used to construct the body of a cycle. The second access point is for continuation of the program after the cycle.

The template "E" is designed to imitate a branching operator construction in an imperative programming language. This template has one access point in the input interface, consisting of the begin and end places of the template. The template has three access points in the output

interface, each consisting of a pair of places, representing branches “then”, “else” and continuation of the program after the branching is done.
Templates “F_1, F_2, F_3, F_4” represent parts of the switch semantic construction of an imperative programming language: the template “F_1” aggregates the construction and contains the begin and the end of it, other templates are representing: completion in *default* case - the template “F_2”, continue after the break operator - the template “F_3” and continue without the operator break - the template “F_4”. Main template “F_1” has one access point of a pair of places in the input interface and three access points in the output interface, designed accordingly to continue switch construction, to build the body of the switch first execution case and to continue program after the switch operator. Access point to continue the construction of the switch has three places, and two other access points have two each. The template “F_2” has a single input access point that consists of three places and no output access points, so it is used as the terminator in switch construction, since after merge with the “F2” template, adding new cases would be impossible. The templates “F3” and “F4” are intended to add new cases to the switch construction and differ from each other by having or not the break operator. With a single access point consisting of three places in the input interface, these templates can be merged only with templates from the switch construction set. The output interface of these templates consists of two access points: the first point has three places and is designed for further development of the switch construction, the second access point has two places and is used for building the body of this case. Thus, to model the behavior of the program in the switch construction, it is necessary to use the aggregator template, merge it step by step with the necessary number of cases and finish the construction with the terminator template.

The template “G” is called a linear section and designed to imitate a simple mathematical expression in an imperative program.

The template “H” is called a terminator. This template has only one access point in the input interface that consist of a pair of places, so after merging with the function body the resulting net the number of unsatisfied access point would decrease.

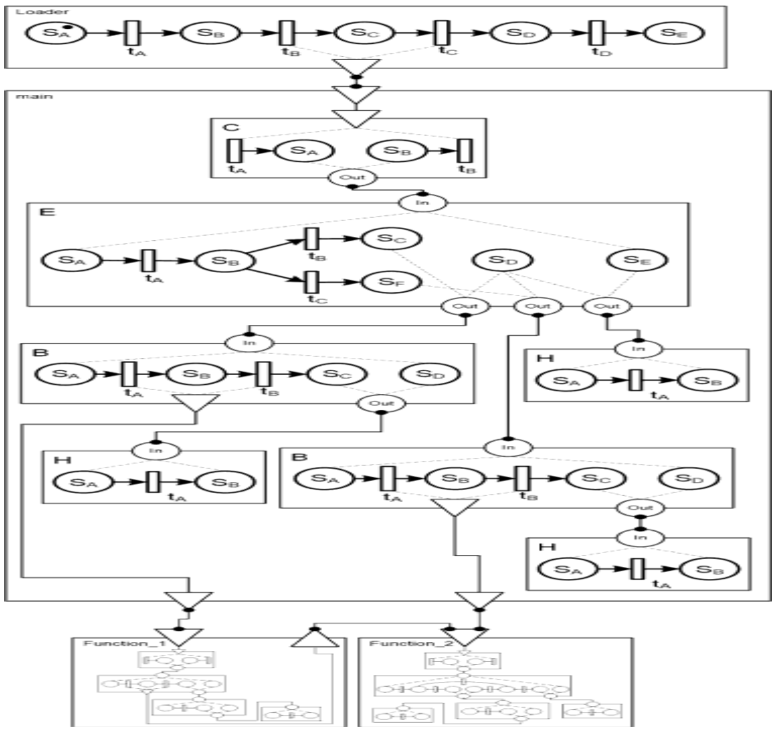


Fig 4. Example of a Petri net generation.

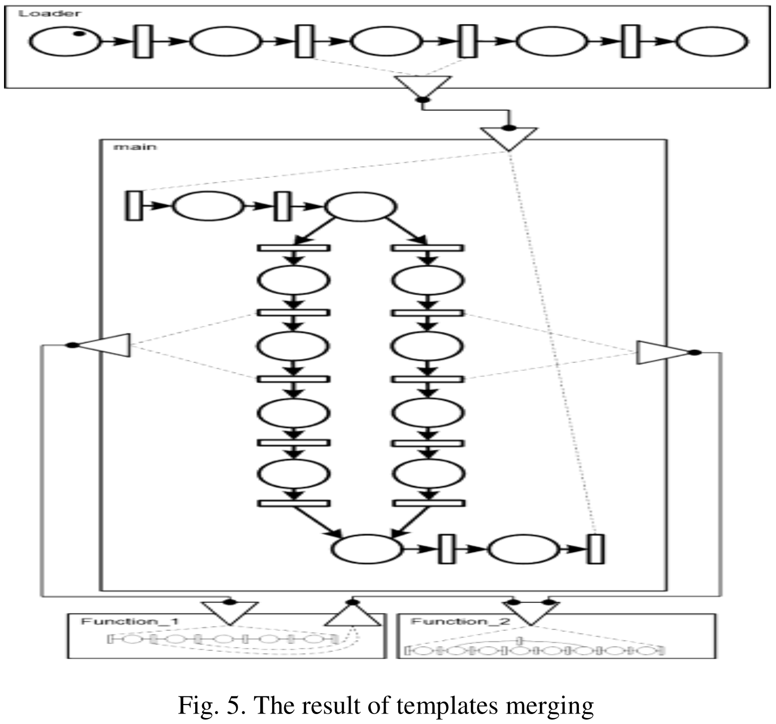


Fig. 5. The result of templates merging

4. EXAMPLE OF A COMPOSITIONAL NET CONSTRUCTION

Let us consider the process of a Petri net modeling an imperative program generation using the templates set given above. In the example of the Petri net generation, next templates were used: loader, function call, function body, branch operator, and terminator. Fig. 4 shows the diagram of the net construction, using drawing conventions listed below:

Big rectangles with a name in the upper left corner represent used templates or resulting function model net. Lines between the access points by places indicate operations of Petri Net composition.

All circles of input simple access points are placed on top of rectangles, and the output on the bottom. Therefore, the order of templates merging coincides with reading rules from left to right, from top to bottom.

Inside rectangles with function names in the upper corner Petri nets modeling function bodies are placed. On the rectangle boundaries, triangles of compositional access points are arranged. Compositional access points can be placed anywhere in rectangle and the direction of merging is specified by the used pair of access points.

Functions “Function_1” and “Function_2” are shown in the example only schematically names of places and transitions in the resulting net are skipped.

Fig. 5 shows the result of templates merging. This net is similar in behavior to a real program consisting of a branch operator, which has two branches in the function “main”. Functions “Function_1” and “Function_2” are called from the first and second branch respectively. In addition, the function “Function_1” calls the function “Function_2”.

5. CONCLUSION

There are only a few works devoted to the generation of program models in term of Petri nets. One should mention some approaches to automatic Petri nets generation based on imperative program source code [7-9]. In the first work, authors construct models from source code to verify shared memory access in a parallel program. GCC was used to get internal program representation and to generate a model of control flow graph. Where only calls to semaphores and mutex are considered. So, the resulting program is generated from model of control flow graph and models of semaphores and other inter-process communication mechanisms. In the second work, authors proposed the number of code patterns to generate a model from the initial program. Each pattern corresponds to some imperative control structure. The Overall model is constructed as directed graph of such templates. In the last work, an approach to generate Colored Petri net of an object-oriented program on Python was considered. However, authors only briefly described how terms of object-oriented language could be represented in terms of Petri nets.

In the presented article, the generation mechanism of Petri nets simulating computational processes constructed from a set of functions in imperative programming languages is considered. Distinguished feature of the described approach is its ability to be implemented in a parallel algorithm. The proposed approach has no direct analogues and is intended to be used in the process of developing and optimizing methods for analyzing models of computational programs in terms of Petri nets that tends to be of great scale, and in particular, for the development of distributed methods of reach ability tree building.

REFERENCES

[1] W. Reisig, “Understanding Pteri Nets. Modeling Techniques, Analysis Methods, Case Studies”, Springer-Verlag: Berlin Heidelberg, 2013, p. 230.

[2] M.-D. Gan, S.-G. Wang, M.-C. Zhou, J. Li, Y. Li, “A survey of reachability trees of unbounded Pteri nets”, Zidonghua Xuebao/Acta Automatica Sinica, 41 (4), pp. 686-693, 2015. DOI: 10.16383/j.aas.2015.c140097.

[3] Y. Du, N. Gu, “Accelerating Reachability Analysis on Petri Net for Mutual Exclusion-Based Deadlock Detection”, Proceedings of 3rd International Symposium on Computing and Networking, CANDAR 2015, pp. 75-81, DOI: 10.1109/CANDAR.2015.65.

[4] H. Ouni, C.A. Abid, B. Zouari, A distributed state space for modular Petri nets, Proceedings of 2015 7th International Conference on Modelling, Identification and Control, ICMIC 2015, pp. 13-18, DOI: 10.1109/ICMIC.2015.7409394.

[5] L.M. Hillah, F. Kordon, “Petri nets repository: A tool to benchmark and debug Petri Net tools”, Lecture Notes in Computer Science, Volume 10258, pp. 125-135.

[6] D.I. Kharitonov, E.A. Golenkov, E.A., Tarasov G.V., Leontyev D.V., “A Method of Sample Models of Program construction in Terms of Petri Nets”, Modeling and Analysis of Information Systems, vol. 22, issue 4 (2015), pp. 563-577.

[7] J.-B. Voron, F. Kordon, “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, SIMTOOLS’08, pp. 1-10.

[8] M. Westergaard, “Verifying Parallel Algorithms and Programs Using Coloured Petri Nets”, Transaction on Petri Nets and Other Models Of Concurrency VI, pp. 146-168, 2012. DOI: 10.1007/978-3-642-35179-2_7.

[9] A. Dedova, L. Petrucci, “From Code to Coloured Petri Nets: Modelling Guidelines”, Transaction on Petri Nets and Other Models Of Concurrency VIII, pp. 71-88, 2013. DOI: 10.1007/978-3-642-40465-8_4.