DOI: 10.15514/ISPRAS-2019-31(4)-11

Computing Transition Priorities for Live Petri Nets

K.G. Serebrennikov, ORCID: 0000-0002-8420-9826 <cyrilsilver94@gmail.com>
National Research University Higher School of Economics,
20, Myasnitskaya st., Moscow, 101000, Russia

Abstract. In this paper, we propose an approach to implementation of the algorithm for computing transition priorities for live Petri nets. Priorities are a form of constraints which can be imposed to ensure liveness and boundedness of a Petri net model. These properties are highly desirable in analysis of different types of systems, ranging from business processes systems to embedded systems. The need for them is imposed by resource limitations of real-life systems. The algorithm for computing transition priorities considered in the study has exponential time complexity, since it is based on construction and traversal of the coverability graph. However, its performance may be sufficient for the majority of real-life cases. This paper covers the design considerations of the implementation, including the approach to handling the high time complexity of the algorithm and optimizations introduced in the original algorithm. We target parallelization as the main method of performance increase. While, for some steps of the algorithm the parallelization approach proves to be viable, for others its applicability is questioned. Analysis of different design decisions is provided in the text. On the basis of the actual implementation an application for computing priorities was developed. It can be used for further analysis of the algorithm applicability for real-life cases.

Keywords: formal methods; Petri nets; coverability graph; priority relation; cyclic behavior

For citation: Serebrennikov K.G. Computing transition priorities for life Petri nets. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 4, 2019. pp. 163-174. DOI: 10.15514/ISPRAS-2019-31(4)-11

Acknowledgments. This work is supported by the Basic Research Program at the National Research University Higher School of Economics.

Вычисление приоритетов срабатывания переходов для живых сетей Петри

К.Г. Серебренников, ORCID: 0000-0002-8420-9826 <cyrilsilver94@gmail.com>
Национальный исследовательский университет «Высшая школа экономики»,
101000, Россия, г. Москва, ул. Мясницкая, д. 20

Аннотация. В данной статье представлен подход к реализации алгоритма вычисления приоритетов срабатывания переходов для живых сетей Петри. Приоритеты являются одной из форм условий срабатывания и могут быть присвоены переходам для обеспечения живости и ограниченности сети Петри. Наличие этих свойств крайне желательно при анализе различных систем, начиная от бизнес-процессов и заканчивая встраиваемыми системами. Необходимость в них обусловлена ограниченностью ресурсов, характеризующей большинство систем из реальной практики. Рассматриваемый в данном исследовании алгоритм для вычисления приоритетов срабатывания переходов имеет экспоненциальную временную сложность, так как основан на процедурах построения и обхода графа покрытия. Однако, его производительность может быть достаточна для большинства практических целей. В данной работе затронуты различные аспекты проектирования реализации, включая подход к решению проблемы высокой сложности алгоритма и примененные к нему оптимизации. В качестве основного метода повышения производительности алгоритма были выбраны параллельные вычисления. Не смотря на то, что для одних шагов алгоритма данный подход продемонстрировал свою жизнеспособность, для других его эффективность оказалась не столь однозначной. В работе представлен анализ различных решений. Также, на основе реализации алгоритма было разработано приложение для вычисления приоритетов срабатывания. Данное приложение может быть использовано для дальнейших исследований реальной применимости алгоритма.

Ключевые слова: формальные методы; сети Петри; граф покрытия; отношение приоритетов; цикличное поведение

Для цитирования: Серебренников К.Г. Вычисление приоритетов срабатывания переходов для живых сетей Петри. Труды ИСП РАН, том 31, вып. 4, 2019 г., стр. 163-174 (на английском языке). DOI: 10.15514/ISPRAS-2019-31(4)-11

Благодарности. Работа выполнена при поддержке Программы фундаментальных исследований Национального исследовательского университета «Высшая школа экономики».

1. Introduction

Petri nets are widely applicable for modeling and analysis of various distributed systems ranging from business processes systems to biological systems. Regardless the nature of such systems their models always have some properties of liveness and boundedness. Properties of liveness include reiteration of all subprocesses and return to some initial state of the system. Properties of boundedness are those related to finiteness of the set of possible states.

In most of the cases, it is highly desirable for the system to have finite set of states, i.e. its model should be bounded. Let us consider, for example, a Petri net shown in fig. 1. This Petri net is a model of a simple producer/consumer system, where the left cycle represents a producer, the right cycle – a consumer, and the place p_3 between them is a buffer. This net is live, i.e. in every reachable marking each transition can eventually fire. The net is unbounded, since the number of tokens in p_3 can be arbitrary large. It means that the buffer overflow will eventually occur. Thus, it is desirable to transform the model into live and bounded preserving the original structure of the net.

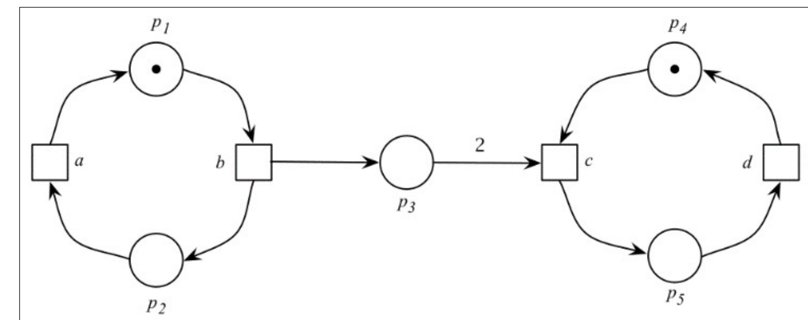

Fig. 1. Example of a marked Petri net. Model of a producer/consumer system

The problem of transformation of a given live and unbounded Petri net into live and bounded without modification of its original structure was considered in [1] and [2]. The authors focused on two approaches to control of Petri net behavior: through priority-based and through time-based constraints on transition firings. Algorithms for computing priorities and time intervals were proposed by them. This paper continues their study.

The proposed algorithms are based on construction of the spine tree, which is a subgraph of the reachability tree, containing exactly all feasible cyclic runs in a net. It represents the behavior that should be saved in a transformed Petri net to preserve the liveness property of the original net. The procedure of obtaining those cyclic runs each of which contains all transitions and is reachable from the initial marking was introduced in [3]. It is based on construction of the coverability graph that is finite by definition but can be extremely large. This fact affects negatively the overall time complexity of the algorithms.

Nevertheless, the performance of these algorithms may be optimal for the majority of real life cases. In this paper we target implementation considerations of the above mentioned algorithms. The study is focused on computation of transition priorities leaving apart computation of time intervals, since these two procedures have common foundation. The main contributions of this paper are the following.

1) An approach to implementation design of the algorithm for computing transition priorities based on construction steps optimization and adoption of parallelization.
2) A brief analysis of the actual implementation coded in Java programming language.
3) A Java application with GUI built upon the algorithm implementation that can be used for running experiments and researching the applicability of the algorithm in practical cases. The application supports the Petri Net Markup Language (.pnml) file format and can be used along with other tools for Petri net modeling and analysis.

The source code of the application along with build and run instructions can be found in the repository[1].

The structure of the paper is as follows. In Section 2 the main theoretical preliminaries are provided. Section 3 contains a brief description of the algorithm under consideration. In section 4 the approach to implementation design is described and results of implementation analysis are presented. Section 5 concludes the paper.

2. Preliminaries

For a more detailed introduction to the concepts presented in this section see, e.g., [4].

Let $\mathbb{N}$ denote the set of natural numbers (including 0). We define a marked Petri net as a tuple

$$(P, T, pre: T \times P \longrightarrow \mathbb{N}, post: T \times P \longrightarrow \mathbb{N}, m0: P \longrightarrow \mathbb{N})$$

where P and T are finite disjoint sets of places and transitions, respectively. $pre(t,p)$ is a number of tokens required to present on place p to enable transition t. The firing of t adds $post(t,p) - pre(t,p)$ tokens to p. Graphically, places are denoted by circles and transitions by squares. There is a directed arc from p to t if $pre(t,p) > 0$. The arc is annotated with $pre(t,p)$ if $pre(t,p) > 1$. Similarly, there is a directed arc from t to p if $post(t,p) > 0$. It is annotated with $post(t,p)$ if $post(t,p) > 1$. The pre-set of a transition t is the set of places p satisfying $pre(t,p) > 0$. The post-set of t is a set of places p satisfying $post(t,p) > 0$. A marking of a net is a mapping $m: P \longrightarrow \mathbb{N}$. The initial marking m_0 is represented by $m_0 0(p)$ tokens on place p.

An initial run is a sequence of transition firings, starting with the initial marking. Reachable markings are all those markings, which can be reached by the initial run. A cyclic run is a finite run starting and ending at the same marking.

A reachability graph of a Petri net is a labeled directed graph, in which vertices correspond to reachable markings of the net. A directed edge from vertex v to vertex v' is labeled with transition t, which is enabled by marking m represented by v and leads to marking m' represented by v'.

A Petri net is bounded if, for each place p, the number of tokens on p does not exceed some fixed bound $k \in \mathbb{N}$, i.e. for each reachable marking m the following is true: $m(p) \leq k$. Thus, a Petri net is bounded if and only if its reachability graph is finite.

A marking m' strictly covers a marking m if and only if for each place $p \in P, m'(p) \geq m(p)$ and $m' \neq m$.

In case of unbounded nets coverability graphs provide finite information about behavior. The construction of coverability graph is based on the notion of the generalized marking, which is formally a mapping: $P \longrightarrow \mathbb{N} \cup \{\omega\}$, where ω denotes an arbitrary number of tokens on a place. A coverability graph is defined constructively: it is constructed successively like the reachability graph starting from the initial marking. However, in case of the coverability graph, when a marking m' represented by a current leave v' in the reachability graph strictly covers a marking m represented by a vertex v, lying on the path from the root to v', then in the coverability graph the vertex v' gets a marking m_w, where $m_w(p) = \omega$ if $m'(p) > m(p)$, and $m_w(p) = m'(p)$ if $m'(p) = m(p)$. Fig. 2 shows a coverability graph of the example shown in fig. 1.

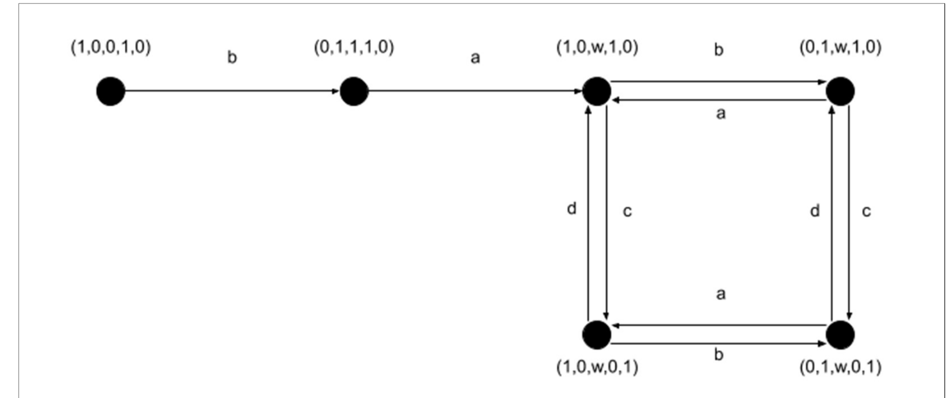

Fig. 2. A coverability graph of the Petri net of fig. 1

The coverability net of a Petri net $N = (P, T, pre, post, m_0)$ is a new Petri net $N' = (P', T', pre', post', m_0')$, which is constructed on the basis of a coverability graph (V, E, v_0) of the original net. The transitions of N' are mapped to the transitions of N by a labeling function $\lambda': T' \longrightarrow T$. The coverability net is formally defined as:

- $P' = V$,
- $T' = E$,
- $pre'((v', t, v''), v) = 1 \, if \, v = v'$,
- $pre'((v', t, v''), v) = 0 \, if \, v \neq v'$,
- $post'((v', t, v''), v) = 1 \, if \, v = v''$,
- $post'((v', t, v''), v) = 0 \, if \, v \neq v''$,
- $m_0'(v) = 1 \, if \, v = v_0$,
- $m_0'(v) = 0 \, if \, v \neq v_0$,
- $\lambda'(v, t, v') = t$.

The extended coverability net is the coverability net, that contains additional places to capture the token count change for ω-marked places of the original net. For each unbounded place p in the original net a place p is added to the extended coverability net. If transition t is in the pre-set of p in the original net N, then all transitions $t' \in T'$ with $\lambda'(t') = t$ are in the pre-set of the added place p. Arc weights are taken into account. The same holds for the post-sets of added places. The initial marking of added places coincides with the initial marking of these places in the original net. Fig. 3 demonstrates the extended coverability net constructed upon the coverability graph shown in fig. 2.

[1] https://github.com/molassar/PN-transition-priority-computer

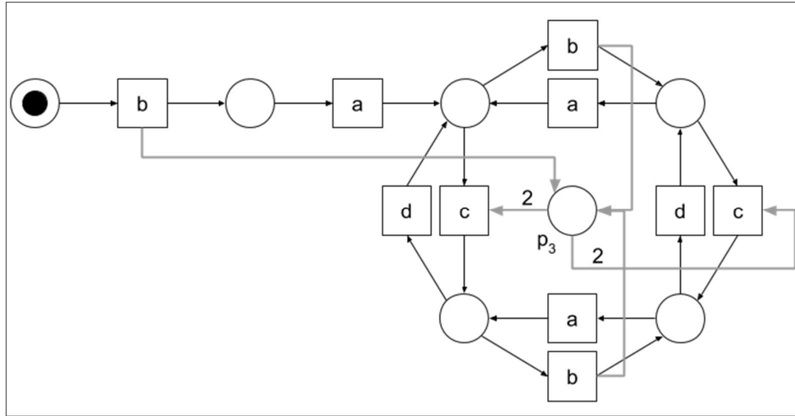

Fig. 3. The extended coverability net of the Petri net of Fig. 1

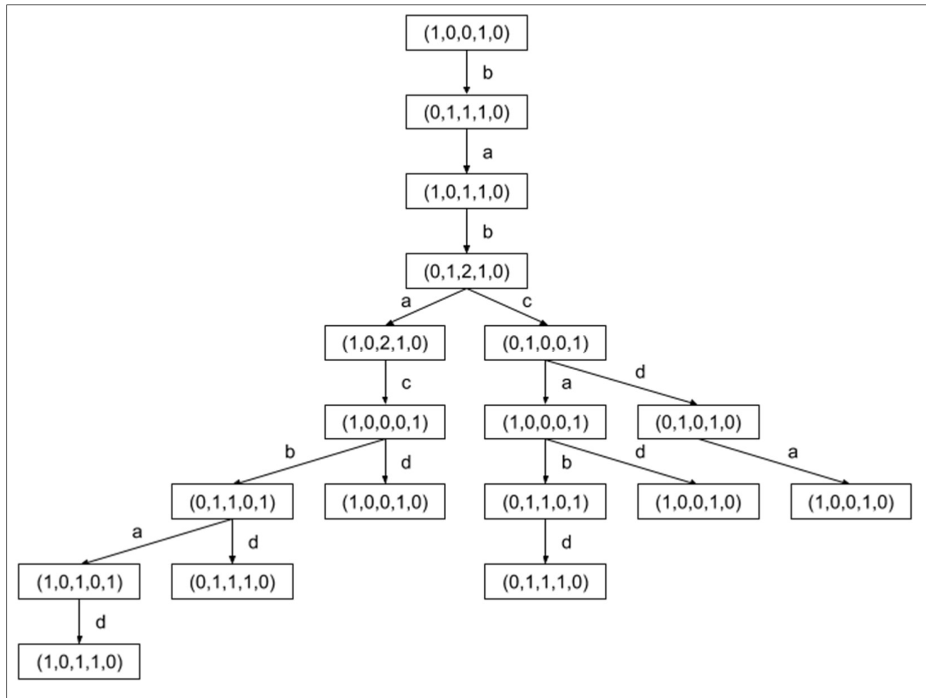

Fig. 4. The spine tree of the Petri net of Fig. 1

We define the set of all minimal feasible cyclic runs together with prefixes leading to the cycles in a Petri net N as

$$C(N) = \{\tau\sigma \mid \tau\sigma * \text{ is an initial run in } N, \\ \tau \text{ does not include } \sigma \text{ and } \sigma \text{ includes all transitions in } N\}$$

where σ is a minimal feasible cyclic run in N and τ is a finite initial run leading to σ. A spine tree is a subgraph of a reachability tree, that contains exactly all runs from $C(N)$. The spine tree contains

the behavior that should be saved in course of transformation in order to keep a Petri net live. For the Petri net in fig. 1 $C(N) = \{babacd, babcad, babcda, babacbd, babcabd, babacbad\}$. Thus, the spine tree of the net has the construction as shown in fig. 4.

A priority relation for a Petri net N is a partial order $(T, \ll)$, i.e., the relation is reflexive, antisymmetric and transitive. A priority relation $\ll$ can be specified by assigning a priority label $\pi(t) \in \mathbb{N}$ to each transition t. Thus, $t \ll t'$ if and only if $\pi(t) < \pi(t')$. A Petri net with priorities is a Petri net together with a priority relation. In a Petri net with priorities if Q is a set of all transitions enabled in a marking m, then only transitions with the highest priority may fire.

3. Algorithm overview

Let N be a live and unbounded Petri net. The task is to check, if it is possible to transform this net into live and bounded by adding priorities to its transitions. To accomplish the task we should find those transition priorities, which exclude runs leading to unboundedness.

It is possible to distinguish two major stages in the algorithm. The first is the search for cyclic runs in a Petri net. The presence of cyclic runs is a necessary condition for existence of transition priorities for a given Petri net. On the second stage the spine tree is built with the cyclic runs found on the previous stage forming its skeleton. The whole algorithm has the following sequence of actions.

1) Given a Petri net build its coverability graph;
2) Given the coverability graph constructed on the previous step build a coverability net;
3) Transform the coverability net into an extended coverability net;
4) Find behavioral cycles of the extended coverability net;
5) If the set of cyclic runs computed on the previous step is not empty build a spine tree in which the cyclic runs form the skeleton;
6) Given the spine tree build a spine-based coverability tree in which all leaves are colored in either red or green[2];
7) Traverse the spine-based coverability tree. For each non-red node a with its incoming edge labeled after transition t_1 if there exists a red sibling b with its incoming edge labeled after transition t_2 add (t_1, t_2) to the priority relation;
8) Assign priority labels to the transitions on the basis of the priority relation computed on the previous step.

The steps 1-4 form the first stage of the algorithm – search for cyclic runs. The procedure was introduced in [3]. The second stage – computation of a priority relation through spine tree construction – is represented by the steps 5-7. It was described in [1].

4. Implementation approach

4.1 Cyclic runs search

The problem of the search for cyclic runs in a Petri net can be reduced to the search for cyclic runs in its extended coverability net. The original algorithm for the cyclic runs search is comprised of four steps. We have reduced the number of steps to two by optimizing the construction phases of coverability and extended coverability nets.

The first step is the construction of a coverability graph. This step can not be avoided since the coverability graph is the foundation for the rest of the algorithm. Since the coverability graph can grow exponentially the overall time complexity of the algorithm is also exponential. We have chosen parallelization to target the problem. The implementation of coverability graph construction and traversal steps was designed to be easily parallelized.

[2] For the complete algorithm see [1].

Listing 1 demonstrates the pseudocode of the algorithm for building the coverability graph. Each node is processed in the following way: the set of transitions is filtered to find the transitions enabled by the marking corresponding to a node, then all the filtered transitions are fired to produce new generalized markings. Directed arcs are added from the vertex labeled by the marking of the node to the vertices labeled by the produced generalized markings. New nodes are generated from the obtained markings for further processing. Processing of a node does not depend on processing of other nodes so this task can be scheduled in parallel. The actual implementation of the pseudocode has the time complexity

$$O(|V| * |T| * d(CG) * |P|)$$

where $|V|$ is the number of vertices in a coverability graph, $|T|$ – the cardinality of the transition set of a Petri net, $d(CG)$ – depth of the coverability graph, i.e., the distance from the root to the most distant vertex, and $|P|$ – the cardinality of the set of places.

```
procedure buildCG(m0,T)
Input:  Initial marking m0 ∈ M,
          M - set of generalized markings;
        set of transitions T
Output:   Directed graph G = (V,E), |V| = |M|
G = initGraph()
root = Node(marking: m0, parentNode: null)
Q = makequeue(root)
while Q is not empty:
   node = dequeue(Q)
   m = marking(node)
   v = Vertex(label: label(m))
   incidentFrom = listIncidentFrom(G,v)
   if incidentFrom is empty:
      fireableT = filter(T, predicate: isFireableFrom(m))
      for each t ∈ fireableT:
         mn = fire(m,t)
         parentNode = node
         while parentNode is not null:
            mp = marking(parentNode)
            if isStrictlyCoveredBy(mn,mp):
               mn = generalize(mn,mp)
               break
            parentNode = parentNode(parentNode)
         u = Vertex(label: label(mn))
         e = Edge(label: label(t))
         addIncidentFrom(G,v,e,u)
         newNode = Node(marking: mn, parentNode: node)
         enqueue(Q,newNode)
```

Listing. 1. Pseudocode of the algorithm for building the coverability graph

We have made two implementations of the algorithm: single-threaded and parallel. In the parallel version, node processing tasks are submitted to a thread pool. Since the worker threads process the submitted tasks asynchronously, there is a need for some synchronization mechanism in order to prevent the function call in the master thread from returning before the graph is fully constructed. For the purposes of synchronization, we have used Phaser. It is a reusable synchronization barrier, which is provided by the standard Java library. The two main operations of Phaser are register and arrive. It is possible to block on Phaser, while the number of registered is not equal to the number of arrived. Phaser is incorporated into the implementation in the following way: the master thread submits the first task to the pool and blocks on Phaser, when a working thread generates a new task it is registered in Phaser, and after completion it arrives. Thus, the function call does not return until all the submitted tasks are processed, i.e. the graph is completely built. We used ForkJoinPool from

the fork/join Java framework as the implementation of the thread pool, since it provides the work-stealing mechanism. If a worker thread runs out of tasks in its pool, it can steal tasks from other threads that are busy. Thus, all the available processing powers are utilized and the performance increases.

We have also conducted a benchmarking of the two implementations with the use of JMH open source tool. Coverability graph construction for the Petri net in Fig. 1 was benchmarked. The *single shot time* mode was selected for the test. In this mode the time for a single operation is measured. Thus the “cold” performance of an algorithm is estimated, since no preliminary warm-up is conducted. This mode is most similar to the real-life usage scenario of the algorithm. The results are presented in Table 1.

Table 1. Benchmark results for implementations of the coverability graph construction algorithm

Benchmark mode	Single-Threaded Score	Parallel Score
Single shot time	5080.768 us/op	7632.496 us/op

The results in Table. 1 demonstrate that the single-threaded implementation performs slightly better for the Petri net of fig. 1. The probable reason is the low complexity of the net. The time complexity of processing a single node is

$$O(|T| * d(CG) * |P|)$$

and, hence, there are strong reasons to believe, that with increase in concurrency of the model and in number of transitions and places the parallel implementation will outperform the single-threaded one. Further experiments conducted with more complex nets proved this assumption.

The construction of the coverability net and the extended coverability net was optimized: we used coverability graph built on the previous stage and a separate data structure for capturing the number of tokens on unbounded places to find all feasible cyclic runs.

Initially, we have designed an implementation of the algorithm for the cyclic runs search, which also targeted parallelization. That implementation resembled closely the implementation of the coverability graph construction algorithm. Both of them were built upon separate nodes processing and since this procedure is completely independent for each node it can be easily parallelized. However, several tests proved this approach to be practically inapplicable. The reason is high memory space consumption. The number of enqueued nodes (logically they represent paths to be checked for cyclic behavior) increases exponentially rapidly exhausting memory capacity.

Thus, we have designed another implementation based on the backtracking principle. It is much more memory efficient, since only one node (path) is processed and stored in memory in every single moment of time. The problem with this approach is that it is difficult to be parallelized. And the developed implementation is also single-threaded. However, processing of each path in the actual implementation has the time complexity $O(|E|)$ where $|E|$ is the number of edges in the coverability graph. This means that in general case, the amount of work needed to be done to process a single path is insignificant for a modern process, and hence no major increase in performance should be observed with addition of parallelization. Nevertheless, the transformation of the proposed single-threaded implementation into a parallel one can be a subject of future research. The pseudocode of the cyclic runs search procedure based on backtracking is presented in Listing 2.

```
procedure backtrackingCyclicRunsSearch(G,m0,upm0,T)
Input:  Coverability graph G;
        initial marking m0 ∈ M,
          M - set of generalized markings;
        initial marking of unbounded places upm0;
        set of transitions T
Output: List of all cyclic runs L
L = initEmptyList()
root = Node(marking: m0, unboundedPlaces: upm0,
```

```
      transition: null, parentNode: null)
path = root
isBacktracked = false
while path is not null:
   if containsCyclicRun(path):
      cyclicRun = extractCyclicRun(path)
      if containsAllTransitions(cyclicRun,T):
         addRun(L,cyclicRun)
         path = parentNode(path)
         isBacktracked = true
         continue
   // no cyclic run was detected
   nodeM = marking(path)
   nodeUPM = unboundedPlaces(path)
   if not isBacktracked:
      incrementPhase(nodeM)
   v = Vertex(label: label(nodeM))
   incidentFrom = listIncidentFrom(G,v)
   isBacktracked = true
   for each (e,u) in incidentFrom:
      tForE = transitionForEdge(e)
      mForU = markingForVertex(u)
      // check if transition tForE from
      //     marking m to marking mForU is not
      //     in the sequence represented by path
      if isNotInSequence(path,tForE,mForU):
         // check if e was not followed already
         // in current phase of nodeM
         if wasNotFollowedInPhase(e,phase(nodeM)):
         // calculate marking of unbounded places
         upmn = fireForUnbounded(nodeUPM,tForE)
         if for every marking in upmn marking >= 0:
            path = Node(marking: mForU,
                        unboundedPlaces: upmn,
                        transition: tForE,
                        parentNode: path)
            markAsFollowedInPhase(e,phase(nodeM))
            isBacktracked = false
   // current path can not be extended; backtracking
   if isBackTracked:
      for each (e,u) in incidentFrom:
         if wasFollowedInPhase(e,phase(nodeM)):
            unmarkAsFollowedInPhase(e,phase(nodeM))
      decrementPhase(nodeM)
      path = parentNode(path)
```

Listing 2. Pseudocode of the backtracking-based procedure for cyclic runs search

A comment on the pseudocode in Listing 2 should be provided. Since it is possible to get into a vertex in multiple different ways, and every time each edge incident from it and not in path should be checked, we have introduced the notion of phase to take account of visited edges and prevent an infinite traversal.

4.2 Priorities computation

The rest of the algorithm is based on the construction of the spine tree of a Petri net and its traversal. The procedure of constructing the spine tree from the list of cyclic runs is quite straightforward and we omit its description here. Similarly, the algorithm for the construction of the spine-based coverability tree was described in details in [1] and we have mostly followed this description in the

process of implementation design. In the spine-based coverability tree the red leaves (ω-leaves) are those nodes which strictly cover some markings preceding them in their branches. To guarantee boundedness they should be cut off. This is achieved with priorities. Listing 3 demonstrates the pseudocode for computing priority relation on the basis of the spine-based coverability tree.

It was proved in [1] that if the relation $\ll$ constructed for the live and unbounded Petri net (N, m_0) is a partial order (i.e. antisymmetric), then $\ll$ is a priority relation for N, and the Petri net $(N, \ll, m_0)$ is live and bounded. However, it is possible for the algorithm in Listing 3 to produce relations with contradictory pairs, thus violating the antisymmetric property. In [2] it is suggested to remove such contradictory pairs from the relation. In this case the Petri net with the resulting priority relation should be checked for boundedness and liveness.

The concluding step of the algorithm implementation is computation of priority labels of transitions. For transitions which are not included in the priority relation the highest priority is assigned since this means that the order of their occurrence is not important. For the rest of transitions a topological sorting is used to order the transitions in the ascending priority and assign a label to each of them with respect to their position. This is possible because the priority relation can be represented as a directed acyclic graph. It should be noted that the obtained priorities can be stronger than it is required since topological sorting does not take into account relative independence of transitions in the priority relation.

```
procedure computePR(treeRoot)
Input:  Root of spine-based coverability tree treeRoot
Output: Priority relation PR = {(t1,t2) : t1 ∈ T ∧ t2 ∈ T},
          T - set of transitions
PR = initEmptyRelation()
Q = makequeue(treeRoot)
while Q is not empty:
   node = dequeue(Q)
   childNodes = children(node)
   redNodes = filter(childNodes, predicate: isRed())
   greenNodes = filter(childNodes, predicate: isGreen())
   yellowNodes = filter(childNodes, predicate: isYellow())
   enqueueAll(Q,yellowNodes)
   for each rn in redNodes:
      // get transition represented by incoming arc of node
      tr = transitionOfIncArc(rn)
      for each yn in yellowNodes:
         ty = transitionOfIncArc(yn)
         addToRelation(PR,(ty,tr))
      for each gn in greenNodes:
         tg = transitionOfIncArc(gn)
         addToRelation(PR,(tg,tr))
```

Listing 3. Pseudocode of the algorithm for priority relation computation

5. Conclusion

In this paper, we have presented our approach to implementation design of the algorithm for computing transition priorities for live Petri nets. In the worst case the performance of the algorithm may be not optimal for the task since it is based on construction and traversal of the coverability graph which can grow exponentially. However, it may prove optimal for the majority of real-life system models. We have proposed parallelization as the main method of handling the complexity. A number of experiments were conducted to estimate its effect on the performance of the implementation. While for some steps this approach proved to be viable, for others its applicability was questioned.

Priority constraints can be helpful in analysis of technical systems, since they ensure liveness and boundedness of such systems. For the purposes of computing priorities the application was developed. This application is based on the algorithm implementation presented in the paper and can be used for further studies on the problem.

However, it should be noted that the application inherits the weak points of the algorithm it is based on, i.e. the high time complexity. Hence, further experiments should be conducted to determine the limits of applicability of the application and the algorithm in particular.

References

[1]. Lomazova I.A., Popova-Zeugmann L. Controlling Petri Net Behavior using Priorities for Transitions. Fundamenta Informaticae, vol. 143, no. 1-2, 2016, pp. 101-112.

[2]. Lomazova I.A., Popova-Zeugmann L., Bartels A. Controlling Boundedness for Live Petri Nets. In Proc. of the 4th International Conference on Control, Decision and Information Technologies, 2017, pp. 236-241.

[3]. Desel J. On Cyclic Behaviour of Unbounded Petri Nets. Application of Concurrency to System Design (ACSD). In Proc. of the 13th International Conference on Application of Concurrency to System Design, 2013, pp. 110-119.

[4]. Reisig W. Understanding Petri Nets. Springer-Verlag Berlin Heidelberg, 2013, 230 p.

[5]. Cormen T., Leiserson C., Rivest R., Stein C. Introduction to Algorithms. The MIT Press, 2009, 1312 p.

[6]. Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D. Java Concurrency in Practice. Addison-Wesley Professional, 2006, 432 p.

Информация об авторах / Information about authors

Кирилл Геннадьевич СЕРЕБРЕННИКОВ в 2019 г. получил степень магистра в области программной инженерии в НИУ ВШЭ, Москва. В число научных интересов входят распределенные системы, формальная верификация распределенных алгоритмов, параллельное программирование.

Kirill Gennadievitch SEREBRENNIKOV received his master degree in software engineering from HSE Moscow in 2019. His research interests include distributed systems, formal verification of distributed algorithms, concurrent programming.