Visualising Data Sets in Structured Occurrence Nets

Talal Alharbi[1,2][0000−0001−8252−9271] and Maciej Koutny[1][0000−0003−4563−1378]

[1] School of Computing, Newcastle University, Newcastle upon Tyne, UK
[2] Faculty of Computer Science and Engineering, University of Hai'l, Hai'l, Saudi Arabia
{talal.alharbi,maciej.koutny}@ncl.ac.uk

Abstract. Visualization techniques have been used with increasing success to assist users in complex system analyses, providing meaningful insights. Structured Occurrence Nets (SONs) originate from occurrence nets (ONs), which are directed acyclic graphs showing causality and concurrency in system executions. Visualization of SONs in tools such as SonCraft is currently limited, as there is a lack of effective support to visualise complex and large data sets. In this paper, we address challenges resulting from the overlapping and out-of-order placement of ONs, and propose a novel solution to deal with this issue based on a step execution policy. Also, we discuss the development of the resulting algorithm, and SonCraft plug-in implementing it.

Keywords: structured occurrence nets · maximal concurrency · visualization

1 Introduction

Complex and large data sets drive considerable attention from information science researchers, crime investigators, and decision makers within governments. Tremendous growth of this kind of data tends to transform its analysis into a hard process. The existence of such data provides important insights, but also raises challenges. In particular, visualization techniques have been used with increasing success to assist users in various analyses, and providing valuable insights about system structure and behaviour [3, 12].

Structured occurrence nets (SONs) [10] originate from occurrence nets (ONs), which are directed acyclic graphs which show the causality and concurrency involved in system executions. Although SONs are supported by visualization in tools such as SonCraft [6–8], there are currently limitations as one cannot effectively visualise data sets automatically. Our aim is to enhance SonCraft making it possible to visualise big data sets in an helpful and understandable way. The approach we propose is to use *semantically driven visualization* which adopts maximal concurrency (a step execution policy in the sense of [1]) to provide structured displays. As a result, the new approach should lead to effective display of complex SONs.

The paper is organised as follows. The next section describes the basic features SONs. Section 3 discusses visualization challenges related to SONs in SonCraft, and proposes a novel solution to deal with them based on maximal concurrency. Section 4 describes implementation and results. The last section contains concluding remarks.

2 Background

2.1 Structure occurrence nets (SONs)

Ocurrence net is an abstract record of a single execution of some computing system in which only information about causality and concurrency between events and visited local states is represented. SONs [6, 11] concept is originally grounded in occurrence nets (ONs), which are directed acyclic graphs which show causality and concurrency of information concerning a single execution of a system [6, 11]. Figure 1 (a) shows an occurrence net (ON). A SON consists of multiple ONs which are associated with each other through various types of formal relationships. SONs are used for recording information concerning the actual behavior of a complex system, and any particular evidence which can be collected in terms of analyzing its past behavior [5]. The most useful way to use SONs is within a detailed analytical investigation, although there is still a lack of investigation support systems which rely on SONs. The significance of SONs results from the fact that their structuring reduces complexity compared to that of any equivalent representation, and they provide a direct means of modelling evolving systems.

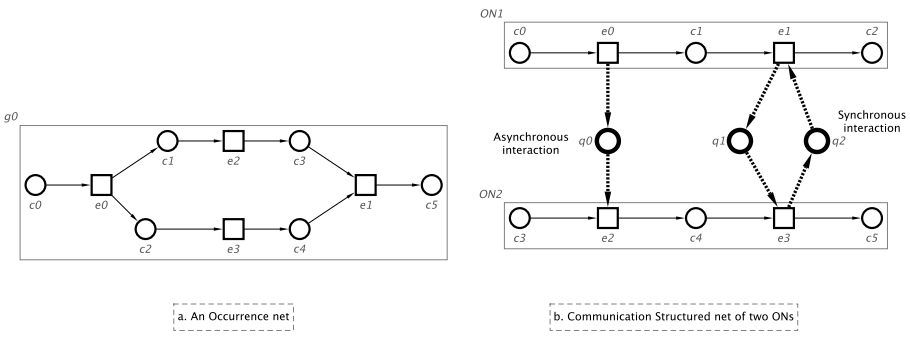

Fig. 1. a. An Occurrence net , b. Communication Structured Net

Communication in (SONs) Communication in SONs is of two types: asynchronous and synchronous [5, 9, 11]. Figure 1 (b) shows a communication structured occurrence net (C-SON) which consists of two occurrence nets, namely $ON1$ and $ON2$. Figure 1 (b) depicts asynchronous communication which is represented through the dashed arc between two events in different ONs, e.g., ($e0$)

and $(e2)$. The second type of communication within SONs is synchronous communication which is represented by the two arcs between events $(e1, e3)$ via the two arcs via channel place ($q1$ and $q2$) [5, 11]. Thus, in any execution consistent with the causal relationships captured by (C-SON), $(e0)$ will never be executed after $(e2)$, although the two events can be executed simultaneously, whereas $(e1)$ and $(e3)$ will always be executed simultaneously.

SONs are underpinned by causal structures which extend causal partial orders with additional ordering, called *weak causality*. Two events ordered in this way can be executed in the order given or simultaneously. Moreover, if two events are weakly ordered in both directions(this means, in particular, that weak causality is not assumed to be acyclic), then they can only be executed simultaneously. In SONs, weak causality results from passing tokens through channel places, whereas the non-channel places introduce the standard causality, as in ONs. In Figure 1 (b), $(e0)$ weakly causes $(e2)$, and the events $(e1)$ and $(e3)$ form weak causality cycle.

Behavioural Structured Occurrence Nets (BSONs) Behavioural structured occurrence nets (BSONs) are used [5] to model the activities of an evolving system. There are two designated levels which represent the execution history. The lower level illustrates behavioral details. The upper level illustrates the evolution of the system [5, 10]. Thus, a BSON provides information on the evolution of a particular individual system. In Figure 2, one level represent what has occurred in terms of the specific system and its evolution, while the other reveals the behaviour of the systems [5].

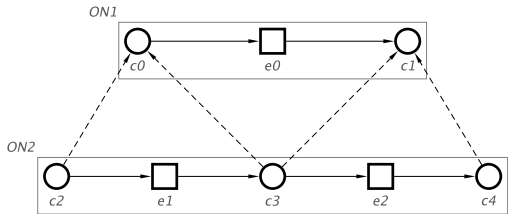

Fig. 2. Behavioural abstraction.

3 Policy Driven Visualization

The current section evaluates SonCraft from the perspective of interaction techniques, as there are issues to be expected when inputting data to SonCraft, after allowing the tool to accept data sets automatically. Such issues include the overlapping of ONs. They can be addressed by using the concept of maximal concurrency, which is a step execution policy in the sense of [1]. Concerning the implementation of new visualization techniques, they deal with the two main purposes of data analysis, namely understanding and exploration. The aim of

exploration in data analysis is to find a perspective which has not been previously discovered. A good example is a crime investigator who searches for behavioural patterns of criminal gangs. As a result, this will allow us to develop SonCraft so that it will be able to help investigators in detecting subtle patterns (modus operandi) embedded inside data automatically retrieved by SonCraft. However, some problems are to be expected, and they are discussed in the next section.

3.1 Loading data sets in SonCraft

Although SonCraft allows visualising data in an attractive way, it does so mainly when cases are manually modelled. The main issue which users face while attempting to load data automatically is overlapping and out-of-order placement. As shown in Figure 3, we should expect both problems after inputting data sets into SonCraft. Figure 4 shows a better way of presenting the same data which

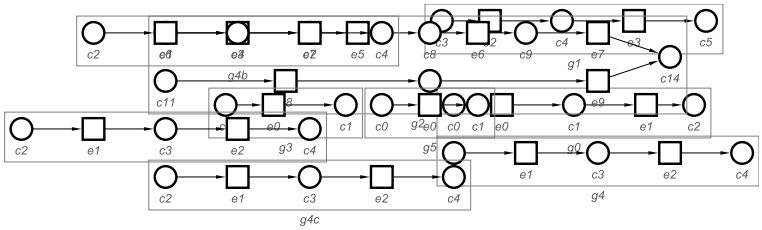

Fig. 3. Overlapping and out-of-order ONs.

avoids overlapping, but still places ONs in an unstructured way. The latter problem will be addressed by applying the idea of maximal concurrency to order the events, and so also the ONs.

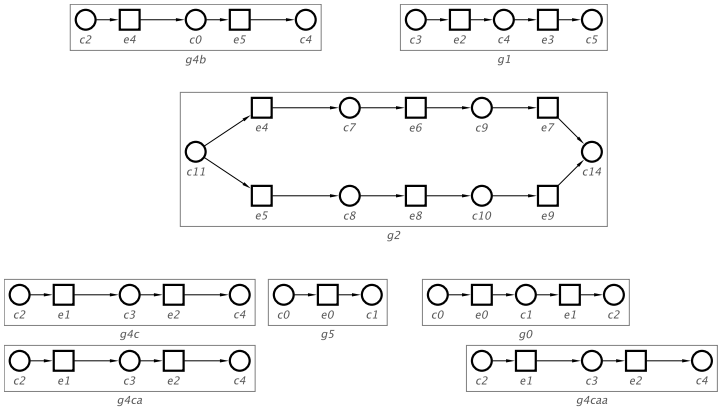

Fig. 4. Removing overlapping of ONs.

There are various reasons due to which overlapping occurs. The first one is due to the fact that although data is loaded automatically when it is retrieved, it is loaded randomly. The other reason is given by the fact that the current SonCraft platform is not prepared to deal with data in an automatic way.

3.2 Proposed solution

Our approach to solving visualisation problems related to SONs consists in a careful consideration of the positioning of the ONs within the workplace area. The method involves the arrangement of ONs in a particular way so as to prevent and avoid overlapping and out-of-order placement by using maximal concurrency (a step execution policy in the sense of [1]). The novel idea is to allow users to automatically control and retrieve data from SonCraft leading to what one might call a *policy-driven visualization*. This approach allows the visualization of concurrency in display systems which are capable of representing it, such as SonCraft.

3.3 Policy-driven visualization algorithm

We have taken advantage of the well-known algorithms such as topological sorting and Tarjan algorithm. The idea behind the algorithm is explained and explanation is supported by clarifying examples. Our consideration to solve the visualization challenge led us to use topological sorting which treats a directed graph as a basis of a linear ordering for that graphs vertices [2]. In order to identify maximally concurrent events in SONs, we needed to find executable events and then co-locate all transitions as maximally concurrent subsets in a particular set X. For example, in Figure 5, there is a maximal concurrency between event (a) in ON $(g0)$ and event (f) in ON $(g1)$, so these events are relocated as one cluster. After placing (a) and (f) in the same maximal 'slice', events (b), (c) and (g) also represent maximally concurrent execution and, similarly, after placing them, events (d) and (e).

The result of applying visualization based on the maximal concurrency policy discussed above is illustrated in Figure 5. As shown, all events within the various ONs are maximal and have been manually clustered. However, our approach is to allow data sets to be retrieved automatically rather than manually. As such, several algorithms have been considered to assist us in doing so. As Khan [2] emphasizes, the main reasoning behind topological algorithms resides in finding a list of start nodes which have no incoming edges [2], and then inserting that node into a particular set X. After that, the node is deleted from the graph and further nodes with no incoming arcs are identified. The development of such an algorithm will help us in finding maximal concurrency events. Thus, our approach consists of using a topological algorithm with the purpose of finding any events which are ready to be executed. In order to identify any event with maximal concurrency, we would need to find an event which has no previous events connected to it. Although SONs are semantically acyclic graphs (in the sense of step sequence semantics) and the topological algorithm was working

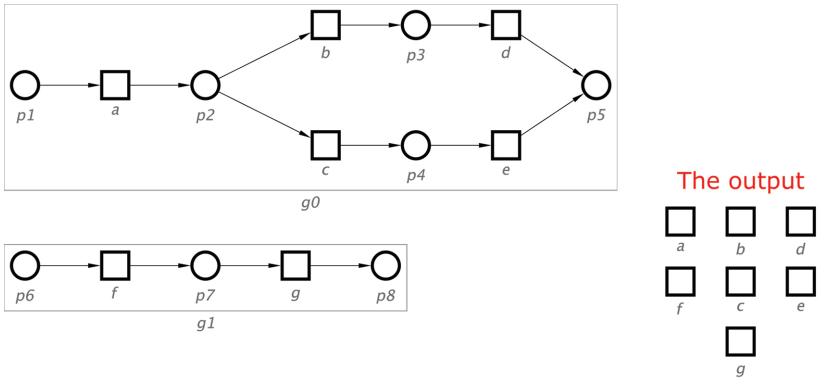

Fig. 5. Two different ONs.

properly, a challenge arose during the process. It was due to a special kind of cycle (weak causality cycle) within SONs, which occurs when two or more different ONs communicate with each other in a synchronous manner, through channel places. Figure 6 illustrates this situation.

As a result, initially, our algorithm could not handle such a situation, because if it was to be applied, for instance, in Figure 6, then event (a) had another event connected with it, i.e., event (d). However, our approach to cope with these problems in such situations can be solved, should they occur, by using Tarjan algorithm. This algorithm was adapted so as to allow it to detect this special kind of cycle (i.e. weak causality cycle). The adaptation implied the ability to push every event of a particular cycle in one set as maximal concurrency cluster, should this type of cycle occur. For instance, after applying the algorithm to the graph in Figure 6, the first cycle between the three existing ONs, namely ($ON2$, $ON3$ and $ON4$) was detected. This was made via channel places ($q0$ and $q1$) and ($q2$ and $q3$). As such, the algorithm will push all events in this cycle, namely events (a, d and f), to one set. After that, event (b) is pushed in one set. Finally, the second cycle between $ON1$ and $ON2$ is detected via channel places ($q4$ and $q5$) in one set, namely events (g and c). Figure 6 illustrates the result. The other interesting case that we have discovered during our investigation of policy-driven visualization in SonCraft is shown in the next example, depicted in Figure 7 (a). Thus, if we apply our algorithm, that use the topological algorithm, we will push events ($e1$ and $e3$) as a set, and then push event ($e2$) as a set, and finally event ($e4$) as another set. Figure 7 (b) shows the result. Although events ($e2$) and ($e4$) can occur together due to asynchronous communication, this could not be handled by the standard algorithm, because event ($e2$) will be treated before event ($e4$). A possible solution is given by an updated algorithm where, after detecting asynchronous communication, one pushes all events within it into the same set. Figure 7 (c) shows the result obtained after applying this solution

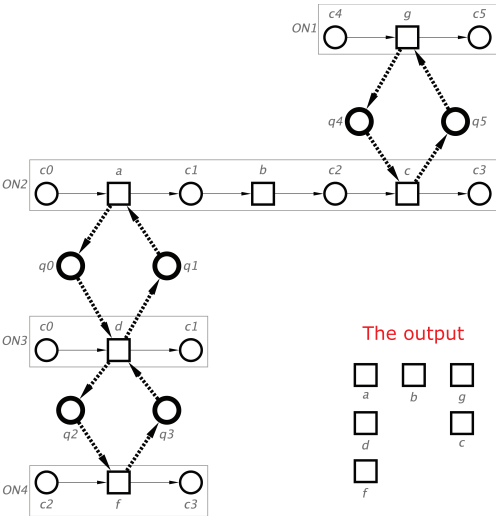

Fig. 6. Special kind of cycle in SONs.

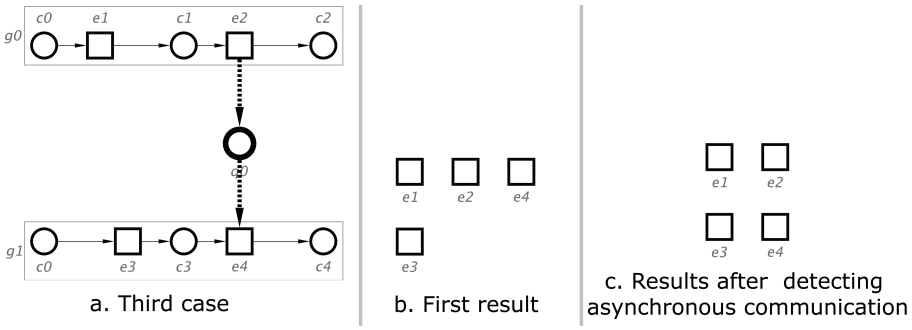

Fig. 7. Third case.

to our example. Note that the algorithm always succeeds when applied to a well-formed SON thanks to the general properties of SONs established in [5].

The pseudo-code of our algorithm for policy-driven visualisation of SONs:

4 Implementation and test results

We have implemented the policy-driven visualization algorithm as a Java plug-in in the SonCraft toolkit (based on the Workcraft platform). Workcraft is a platform that provides a framework for the development and analysis of graph models [6]. We have created a menu (Retrieve Big Data) that has two features. As shown in Figure 8, the first submenu a is 'Import SON data', and the other is 'Maximal ONs events'.

Algorithm 1: Visualize SONs' events By Maximal Concurrency

```
   Input: set_of_ons : Set of ONs ;
   Output: set of maximal sorted ONs' transitions
 1 Start ;
 2 array set_of_ons_to_check = set_of_ons ;
 3 array set_of_maximal_concurrencies = array[array[Node]] ;
 4 array temp_maximal_concurrencies = [] ;
 5 boolean loop_again = true ;
 6 while (loop again) do
 7 |  loop_again = false ;
 8 |  array temp_maximal_concurrencies = [] ;
 9 |  for (ONs ons in set_of_ons_to_check) do
10 |  |  array ons_nodes = ons's nodes;
11 |  |  array ons_set_of_cycle = [];
12 |  |  ons_set_of_cycle = Tarjan(ons);
13 |  |  for (Node node in ons_node) do
14 |  |  |  if (node is a transition) then
15 |  |  |  |  loop_again = true ;
16 |  |  |  |  if (node has no previous transition) then
17 |  |  |  |  |  boolean node_belongs_to_a_cycle;
18 |  |  |  |  |  for (Cycle cycle in ons_set_of_cycles) do
19 |  |  |  |  |  |  node_belongs_to_a_cycle = false;
20 |  |  |  |  |  |  if (node belongs to one of set_of_cycles' cycles)
   |  |  |  |  |  |   then
21 |  |  |  |  |  |  |  push all cycle nodes to
   |  |  |  |  |  |  |   temp_maximal_concurrencies;
22 |  |  |  |  |  |  |  delete all cycle nodes from ons_nodes;
23 |  |  |  |  |  |  |  delete detect cycle from set_of_cycles;
24 |  |  |  |  |  |  |  node_belongs_to_cycle = true;
25 |  |  |  |  |  |  |  break;
26 |  |  |  |  |  |  end
27 |  |  |  |  |  end
28 |  |  |  |  |  if (node belongs to cycle = false) then
29 |  |  |  |  |  |  push node to temp_maximal_concurrencies;
30 |  |  |  |  |  |  delete nodes from ons_nodes;
31 |  |  |  |  |  |  break;
32 |  |  |  |  |  end
33 |  |  |  |  end
34 |  |  |  end
35 |  |  end
36 |  end
37 |  push temp_maximal_concurrencies to set_of_maximal_concurrencies ;
38 end
39 return set_of_maximal_concurrencies  ;
40 End ;
```

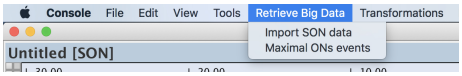

Fig. 8. Retrieve Big Data menu.

Here is a brief explanation of the functionality of our plug-in. Firstly, our algorithm finds the starter conditions. These are conditions without pre-events. For every starter condition, we initialise two arrays. This will track which conditions will be added or removed for the next loop step. Next, we will find all cycles in the graph via the Tarjan Algorithm and store them in a list. Afterwards, we need to find starter events and store them in a list. Starter events are all events that do not have previous events. Next, we will loop through all starter conditions. If we catch any starter events (for every event), we will add them to our temporary array if they have no connections. We will also add them to our temporary array if they have two connections, and the event does not follow an existing event in the temporary array. Then we will set the next condition as a starter and mark it if it is not in a cycle. However, if it is in a cycle, we will mark the starter condition and treat it separately. Next, we will loop through all starter conditions again. For every starter condition, we check to see if the condition is not being marked for removal (the condition has been skipped because the event following the condition is a part of Trajan's cycle). If this is the case, we obtain its post-event and all cycle events. Just after the two loops, we will decide if a set of maximal concurrency has completed or not. Finally, after we are sure that there are no more starter conditions, we will display the maximal concurrency events. [13] We now discuss the results obtained after the

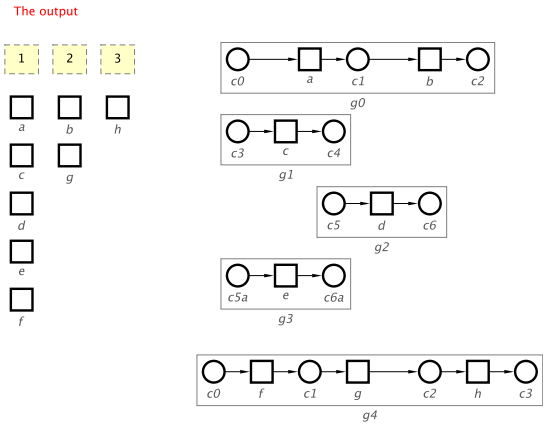

Fig. 9. First test result.

plugin was developed and implemented. The first test was a scenario of different

ONs with no communication between them. This shows that the plugin works correctly. For instance, in Figure 9, the first set comprises events (a, c, d, e and f). Afterwards, we obtain the next set consisting of two events (b and g). The last set has one event (h).

Figure 10 provides another example of how our plugin treats complex scenario. It involves a model with three synchronous communications. As Figure 10 shows, the algorithm is working properly and the three maximal sets are (a, d and f), (b, e and g) and (c).

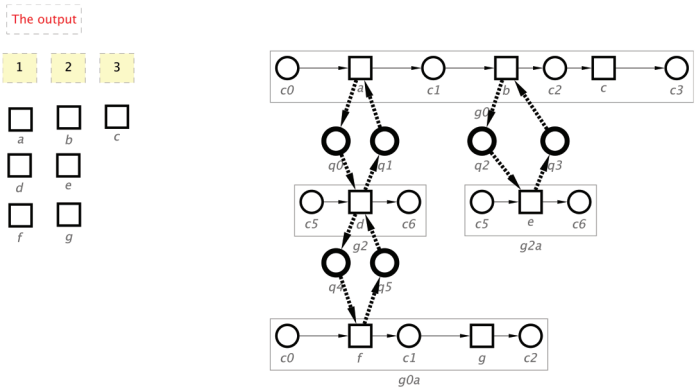

Fig. 10. Second test result.

Figure 11 shows the last test case. It involves asynchronous communications, and the algorithm successfully pushes its events into one set, namely ($e2$ and $e4$).

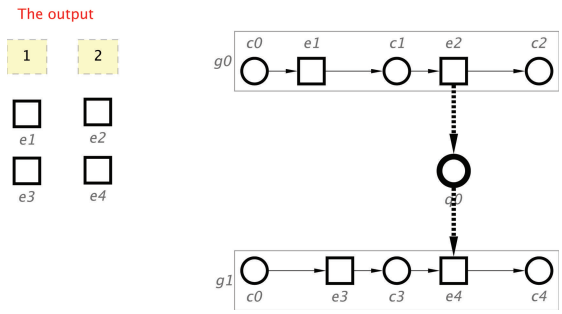

Fig. 11. Third test result.

Finally, as displayed in the next figure 12, we provided a figure that shows a larger example of our plug-in.

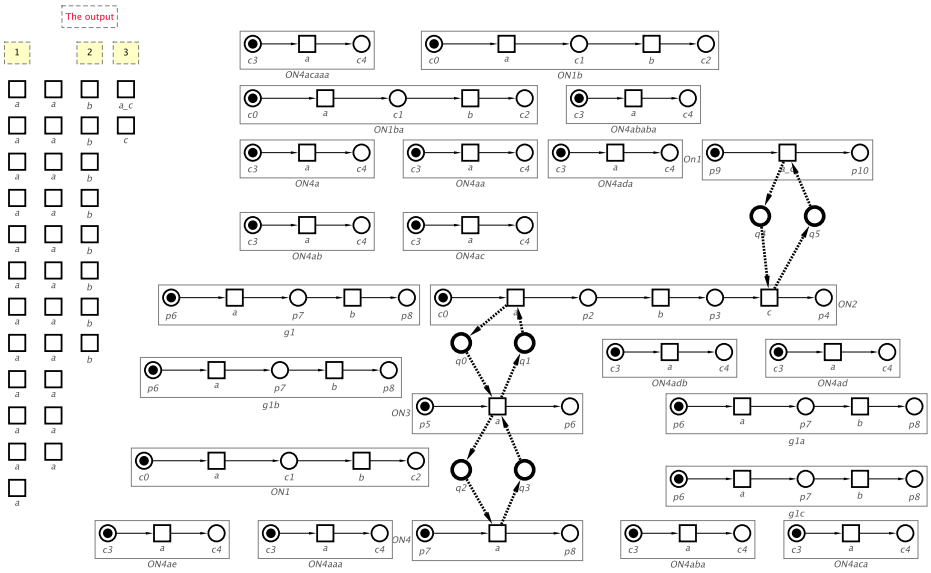

Fig. 12. Fourth test result.

5 Concluding remarks

In this paper, the problems of ONs overlapping and out-of-order placement have been looked into, and a novel solution to address them has been proposed. The design and development of the corresponding algorithm has been discussed. We provided test examples showing that our algorithm works properly.

In future work we will address the visualization of behavioural abstraction through maximal concurrency policy. Another direction for future work is to implement visualization based on local maximal concurrency [4]. It is a step execution policy that introduces an extension of the basic model of Petri nets, namely PT-nets, by adding the notion of located transitions and locally maximally concurrent executions of co-located transitions in order to represent the compartmentation of membrane systems [4]. The main idea depends on specifying the locality for transitions in PT-nets, each transition belonging to a fixed unique locality. The precise mechanism to achieve this is to introduce a partition of the set of all transitions using locality mapping [1].

Acknowledgements

The authors acknowledge financial support provided by University of Hai'l. Also, we would like to thank the reviewers for their insightful comments on the paper.

References

1. Darondeau, P., Koutny, M., Pietkiewicz-Koutny, M., Yakovlev, A.: Synthesis of Nets with Step Firing Policies. Fundam. Inform. 94(3-4), 275-303 (2009)
2. Kahn, A.B.: Topological sorting of large networks. Communications of the ACM 5.11, 558-562 (1962)
3. Keim, D.A.: Information visualization and visual data mining. IEEE Transactions on Visualization and Computer Graphics 8.1, 1-8 (2002)
4. Kleijn, J., Koutny, M., Rozenberg, G.: Towards a Petri net semantics for membrane systems. International Workshop on Membrane Computing. Springer, Berlin, Heidelberg (2005)
5. Koutny, M., Randell, B.: Structured occurrence nets: A formalism for aiding system failure prevention and analysis techniques. Fundamenta Informaticae 97, 41-91 (2009)
6. Li, B., Koutny, M., Randell, B.: SonCraft: A tool for construction, simulation and verification of structured occurrence nets. Tech. Rep. CS-TR-1493, School of Computing Science, Newcastle University (2016)
7. Li, B., Randell, B.: SonCraft user manual. Tech. Rep. CS-TR-1448, School of Computing Science, Newcastle University (2015)
8. Li, B., Koutny, M., Randell, B., Bhattacharyya, A., Alharbi, T.: SonCraft: A Tool for Construction, Simulation and Analysis of Structured Occurrence Nets. to appear in proceedings of ACSD (2018)
9. Randell, B., Koutny, M.: Structured Occurrence Nets: Incomplete, contradictory and uncertain failure evidence. School of Computing Science Technical Report Series (2009)
10. Randell, B.: Occurrence nets then and now: the path to structured occurrence nets. International Conference on Application and Theory of Petri Nets and Concurrency. Springer, Berlin, Heidelberg (2011)
11. Randell, B.: Incremental construction of structured occurrence nets. Computing Science, Newcastle University (2013)
12. Shneiderman, B.: Inventing discovery tools: combining information visualization with data mining. The Craft of Information Visualization, 378-385 (2003)
13. Li B., Koutny M., Randell B.: Workcraft. `https://workcraft.org` (2015)