Structural Induction as a Method to Distribute the Generation of a Trace Language Representation for Complex Systems

Krzysztof Łęcki*, Jerzy Tyszkiewicz, and Jacek Sroka*
Institute of Informatics, University of Warsaw, Poland

Abstract. The paper presents a method to distribute the generation of all possible bounded length sequences of events for a given Petri net. The crucial benefit of our method is that it permits distributing of the generation process and allows to apply Big data processing techniques. The net is decomposed into functional fragments, each of them is simulated independently to produce the set of its sequences, which are combined to yield the sequences for the whole net. This process can be executed by structural induction, and facilitates re-usability of partial results.
To limit the amount of data transferred during the computation, which is the critical factor, we represent these sequences in a compact form of Mazurkiewicz traces, which identify all sequences which differ by interchanging the order of subsequent independent events. Therefore a single trace corresponds to all distributed runs of the net, where tasks that can be executed concurrently are recorded in all possible orders.
Our tool can be useful in process mining, in particular in conformance checking. In this use case, a Petri net model of a process is compared to its real-life behaviour, recorded in execution logs. As the investigated logs are large, the gain from distributing the computation is desired.

Keywords: process mining, conformance checking, distributed computation, Big data, Spark, Petri net, elementary net system, Mazurkiewicz traces, structural induction

1 Introduction

The focus of this paper is providing a methodology for efficient and distributed generation of the set of all limited length runs of a given Petri net.

Petri net (PN) [4] is a model of a distributed system. The formalism of such nets is a common and convenient modelling language. Their applications areas range from workflow systems, representing decision processes in a business organization, through the design of control structures in concurrent programming, to the models of regulatory networks found in living cells.

One of the possible application areas for our tool is *process mining* [1, 6]. It is a relatively young research area, whose main aim is to develop methods and

* Sponsored by National Science Centre based on DEC-2012/07/D/ST6/02492.

techniques to discover, monitor and modify real processes, making use of the knowledge implicitly present in their event logs. Certainly, achieving such a goal requires methodology to perform (automated) *process discovery*, i.e., constructing process models by mining the event log data. When the model is available, *conformance checking* becomes important — it is the task of investigating and analysing deviations between the real process and its model. The next steps might then be model extension, model repair or case prediction.

The generation of all possible fixed length runs of a model is our contribution towards conformance checking. Assuming that a Petri Net model of the process is already available, it is then natural to compare the sequences of events generated by the model with those indeed recorded in the log. It is relatively easy to do one way: check if the sequences of events present in the log are indeed permitted by the model. The other direction is much harder: to check if there are sequences of events permitted in the model, which do not appear in the event log, and if so, characterize them, estimate their frequency among all runs, discover their common properties, etc. In fact, using our tool to produce the complete set of sequences generated by the model (up to some length), allows one to mine the difference between the model's event log and the real system's event log, providing otherwise unavailable information about what the real system cannot do, as opposed to its model. Existing solutions, like various applications of SAT-solvers, produce single runs, so can provide answers only to very detailed existence questions and do not provide the general picture.

Given a PN N and a limit k our tool produces the representation of all event sequences of length up to k of N. The cardinality of this set grows extremely fast with k, thus our goal was to distribute the computation. A naïve method for exhaustive sequence generation would be to explore the possible state space. Such a method would be difficult to distribute due to excessive communication necessary to prevent repeated exploration of the same trajectories on different machines. The work could be distributed based on the order of events in the simulation, where each machine gets a fragment of the search space, composed of the sequences starting with a particular prefix. Yet, for different prefixes there can be significant difference in the number of possible result sequences, which would introduce skew and even if the computation would be balanced, then common suffixes, or subsequences would lead to replication of work.

Instead we apply the techniques of Mazurkiewicz [3], where the sequences for the net can be obtained by combining sequences for its subnets. This allows us to reduce the problem to iterative Big data processing computation, which can be efficiently distributed with Big data frameworks like Hadoop and Spark. Distributing the computation is done by structural induction on the PN model. This has additional practical consequences. In a net with several identical components it is enough to compute their behaviour once and then reuse it several times. To limit the amount of data transferred during the computation, which is the critical factor, rather than with sequences, we deal with *Mazurkiewicz traces* [3] and exploit the prefix closure property. Each such trace represents a whole set of sequences, which differ only by interchanging the order of subse-

quent transitions that are independent in the net. This reduces the size of the data necessary to store all runs by many times without losing any significant information. For example for sequences of length bounded by 7 we observed average reduction of 29 times. Indeed, every trace represents a set of runs in which subsequent independent events occur in different order or equivalently the trace represents a run where all subsequent independent events occur simultaneously. We further reduced the size of the representation (by 35% for the same example), thanks to the observation of the prefix closure property for the sequences of events. However, even with this additional optimization and using the trace representation, we still end up in the Big data range even for not too large k.

For implementation of our ideas we use Apache Spark engine [7]. The additional advantage of this is that the computed traces are immediately available as Spark's Resilient Distributed Dataset (RDD) and can be further processed/analysed without any overhead for parsing. Such post-processing could include analysing the deviations between the real observed process and the model, or computing some statistics, e.g., in which percent of all possible runs some chosen event always precedes some other event or what is the average delay/distance between some events.

2 Elementary net systems and net languages

We focus on *elementary nets systems* (ENS) [5] which are a kind of PN with a restriction that each place can contain up to one token. That is, for a transition to be enabled not only all its input places need to contain tokens, but also all its output places need to be empty. Formally, ENS is an ordered triple (P, T, F), where P is a finite set of places, T a finite set of transitions and $F \subseteq ((P \times T) \cup (T \times P))$ defines the flow relation. The marking is defined as $M \subseteq P$ and a transition t is considered enabled if (1) $\forall_{p \in {}^\bullet t} p \in M$ and (2) $\forall_{p \in t^\bullet} p \notin M$ where ${}^\bullet t \subseteq P$ ($t^\bullet \subseteq P$) is the set of input (output) places of t. An enabled transition can fire, which changes the state from M to M' denoted as $M \xrightarrow{t} M'$ where $M' = ((M \setminus {}^\bullet t) \cup t^\bullet)$. For technical reasons we assume that the nets with which we deal do not include immediate loops, that is for every transition t it holds that ${}^\bullet t \cap t^\bullet = \emptyset$.

An ENS with an initial state defines a transition system whose size can be exponential with respect to the number of places. Such a system defines a language of sequences (strings) with symbols from T — its net language — that can be observed as the outcomes of possible runs of the system starting in the initial marking. Formally, for $N = (P, T, F)$ and for some initial marking M_0, we say that sequence/string $w = w_1 w_2 w_3 \dots w_n$ for $w_i \in T$ is in the net language of N if $M_0 \xrightarrow{w_1} M_1 \xrightarrow{w_2} M_2 \xrightarrow{w_3} \dots \xrightarrow{w_n} M_n$ for some markings $M_1, \dots, M_n$.

3 Language synchronization

Synchronization of net languages [3] corresponds to composition of nets. We use the synchronization operation on net languages to distribute net language

computation for complex nets by decomposing the net and then synchronizing net languages of the obtained components.

We consider *composition* for nets that have no common places but may have common transitions. During this operation such transitions form an interface on which the composition is conducted and are merged together. The marking of the composite net is obtained as the sum of the markings of the composed nets.

Composition of sequence of non-trivial nets $N_1, N_2, \ldots, N_n$, where $N_i = (P_i, T_i, F_i)$ and $P_i \cap P_j = \emptyset$ for $i \neq j$, is defined as the net $N_1 + N_2 + \ldots + N_n = (\bigcup P_i, \bigcup T_i, \bigcup F_i)$. An example of the net composition is presented in Figure 1 for $N_1 = (P_1, T_1, F_1)$ and $N_2 = (P_2, T_2, F_2)$ where $P_1 \cap P_2 = \emptyset$ and $T_1 \cap T_2 = \{b, c\} \neq \emptyset$ is the common interface. The resulting net $N_1 + N_2 = N$ is obtained by merging the interface transitions of the components.

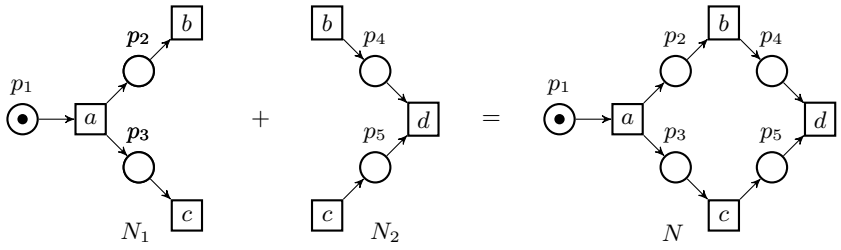

Fig. 1: Net composition example.

Complementarily we can define net *decomposition* of a non-trivial net by specifying partition of its set of places into non-empty sets. Each such set defines a subnet together with transitions adjacent to its places and corresponding edges (transitions can be replicated for multiple subnets). Suppose that we have a net N and its decomposition into $N_1 + N_2 + \ldots + N_n$. The language $L(N)$ — net language of N — can be defined in terms of net languages $L(N_1), L(N_2), \ldots, L(N_n)$. Consider a $seq \in L(N)$. Observe, that each of nets N_i accepts a sequence $\pi_{T_i}(seq)$ where $\pi_A(seq)$ is the projection of sequence seq onto alphabet A. This follows from the observation that if there exists a series of actions seq in system represented by N, then if we look only at places and transitions belonging to the component N_i and obscure remaining parts of the system we get $\pi_{T_i}(seq)$.

Formally, *synchronization* [3] of (arbitrary) languages L_1 over alphabet A_1 and L_2 over A_2 is defined as the language $L_1 \parallel L_2$ over alphabet $A_1 \cup A_2$ such that: $seq \in (L_1 \parallel L_2) \Leftrightarrow \pi_{A_1}(seq) \in L_1 \wedge \pi_{A_2}(seq) \in L_2$.

The smallest element of net decomposition, containing a single place, is called *atom*. Each net is a composition of all its atoms. We can observe that an atom $A = (\{p\}, T, F)$ for a marking $M = \{p\}$ has a net language which is defined by the regular expression: $((p\bullet)(\bullet p))^*(p\bullet + \epsilon)$. This observation allows us to calculate the language of a net by decomposing it into atoms, calculating net languages of each atom and synchronizing those languages incrementally one by one.

We can analyse this using the resulting net from Figure 1. We will calculate net languages up to length 4. Let atom A_i be based on place p_i for $i = 1, 2, 3, 4, 5$ and let the initial markings be given by projections of the marking presented in the figure. We have: $L(A_1) = \{\epsilon, a\}$, $L(A_2) = \{\epsilon, a, ab, aba, abab\}$, $L(A_3) = \{\epsilon, a, ac, aca, acac\}$, $L(A_4) = \{\epsilon, b, bd, bdb, bdbd\}$, $L(A_5) = \{\epsilon, c, cd, cdc, cdcd\}$. It can be observed that the following synchronizations hold: $L_{12} = L(A_1) \parallel L(A_2) = \{\epsilon, a, ab\}$, $L_{123} = L_{12} \parallel L(A_3) = \{\epsilon, a, ab, abc, ac, acb\}$, $L_{1234} = L_{123} \parallel L(A_4) = \{\epsilon, a, ab, abc, abcd, abd, acb, acbd\}$, $L(N) = L_{1234} \parallel L(A_5) = \{\epsilon, a, ab, abc, abcd, acb, acbd\}$.

4 Distributing net language computation

The idea to decompose a net into atoms, compute net languages for them and then generate the net language for the initial net by incrementally synchronizing the languages of atoms gives us means to distribute the net language generation problem.

This procedure is presented in the Listing 1. Its correctness follows from the observation that synchronization is distributive, i.e., $(A \cup B) \parallel C = (A \parallel C) \cup (B \parallel C)$ and $A \parallel (B \cup C) = (A \parallel B) \cup (A \parallel C)$. Thus, to synchronize two languages we can break one of them into a collection of languages containing individual sequences and synchronize all such singletons with the second language and finally take union of the results. This observation forms the basis for how our algorithm distributes work. As the incrementally generated net language is large its subsets (partitions in Spark) can be stored on different nodes in the cluster and can be processed independently.

As the net languages of atoms are easy to generate and relatively small if bounded by reasonable length, we can use the broadcast mechanism available in Spark (and in Hadoop) to make them available on every node of the cluster. In that case the computation could be organized in a loop where in each step we would compute the net language of the net that is obtained from the net from the previous step by composing it with next atom. The net language of the composed net would be computed from the net language of the net from the previous step by synchronizing its every sequence as singleton net language with the net language of the composed atom.

The `synchr(seq, bc.value)` operation performs the synchronization $\{seq\} \parallel bc.value$. It can be implemented by synchronizing the $\{seq\}$ language with each language $\{s\}$ for $s \in bc.value$ and taking set union of the results. The definition of the synchronization operation is nonconstructive. We will present our method of computing synchronization in Section 6. For now let us just note that synchronization of two singleton languages can either contain no nonempty (epsilon) sequences, or contain a single nonempty sequence, or contain multiple nonempty sequences. Thus we have to repartition the RDD after each step, to equalize the sizes of subsets of the resulting sequences stored on individual nodes. This way, in the next iteration of the main loop computation will take comparable amount of time on each of the nodes. Furthermore, we also need to eliminate duplicates,

```
def genLang(n: Net, k: Int, numPart: Int, sc: SparkContext) {
  val atoms = n.splitToAtoms() //split net to atoms
  //initialize with the language for the first atom
  val l = atoms.head().getLanguage(k)
  val res = sc.parallelize(l, numPart)
  //structural induction
  //incremental generation by synchr. with successive atom's languages
  for (nextAtom <- atoms.tail()) {
    l = nextAtom.getLanguage(k)
    var bc = sc.broadcast(l) //broadcast next atom's language
    //synchr. all pairs of sequences from two languages
    res = res.flatMap(seq => synchr(seq, bc.value))
             .map(seq => if (seq.length() > k) seq.substring(0,k)
                         else seq).distinct().repartition(numPart)
    bc.unpersist()
  }
  res.saveAsTextFile(hdfsOutputPath) //store output
}
```

Listing 1: Language generation by incremental synchronization of atom languages.

which will be efficiently done by Spark — Spark can merge both repartitioning and duplicate elimination into one operation.

5 Traces

In the previous section we presented a method to calculate the net language of an ENS in a distributed manner. Now we introduce a technique to limit the amount of data transferred and at the same time reduce the amount of computation.

The net language consists of all possible sequences of events in a net even if they differ only by ordering of subsequent independent transitions. In this section we introduce traces which group all sequences that differ only in ordering of subsequent independent transitions. Traces allow us to represent the same information about the net as sequences but from the concurrent viewpoint — as if all subsequent independent events occur simultaneously instead of considering their linearisation.

First, we formalize independence/dependence of transitions. Following [3] a *dependence relation* is any finite, reflexive and symmetric relation. For a net $N = (P, T, F)$ we define a dependence relation $D_N \subseteq T \times T$ and an *independence relation* $I_N \subseteq T \times T$ over alphabet T based on how firing one transition affects the possibility to fire another one. If firing of one transition enables or disables another transition we call them dependent. On the other hand if transitions are independent firing one of them should not immediately interfere with the ability to fire the other. It is easy to see, that transitions are considered dependent,

if they share a neighbouring place, otherwise they are independent. Formally, $(t_1, t_2) \in D_N$ iff $(\bullet t_1 \cup t_1 \bullet) \cap (\bullet t_2 \cup t_2 \bullet) \neq \emptyset$. Independence relation is defined as the complement of dependence relation $I_N = (T \times T) \setminus D_N$ and we say it is induced by dependence D_N. For given dependence D we mark independence induced by it as I_D. Note that dependence D carries information about its alphabet which is denoted $\Sigma(D)$, similarly we use $\Sigma(I)$ for independence I. It is an easy check that dependence relation defined by us for a net is indeed finite, reflexive and symmetric. For net N from Figure 1 we have $D_N = \{a, b\}^2 \cup \{a, c\}^2 \cup \{b, d\}^2 \cup \{c, d\}^2$, so independence induced by D_N is $I_{D_N} = \{(b, c), (c, b), (a, d), (d, a)\}$.

For an alphabet A and a dependence D over A *trace equivalence* $\equiv_D$ is defined as the least congruence in A^* such that for all $(a, b) \in I_D \Rightarrow ab \equiv_D ba$. Traces are equivalence classes of $\equiv_D$. A trace over dependence D represented by a sequence seq is denoted $[seq]_D$. For a language of sequences L, by $[L]_D$ we denote the language of traces $[L]_D = \{[seq]_D \mid seq \in L\}$. Note that traces over full dependence $A \times A$ are singletons.

For a given net N, $[L(N)]_D$ is called its *trace behaviour* or *trace language*. It is important to note that trace behaviour of a net is *consistent*, i.e., $\{seq \mid [seq]_D \in [L(N)]_D\} = L(N)$.

6 Trace language computation

Without losing any information we can store every trace as one of its representatives, i.e., one of the sequences that are in this equivalence class of the equivalence relation $\equiv_D$. The dependence relation that is needed to calculate the full net language of sequences of events from the representatives of its trace behaviour can be easily calculated by definition from the net.

Synchronization of trace languages T_1 over dependence D_1 and T_2 over D_2 is defined as the trace language $T_1 \parallel T_2$ over $D_1 \cup D_2$ (over the alphabet $\Sigma(D_1) \cup \Sigma(D_2)$) such that $t \in (T_1 \parallel T_2) \Leftrightarrow \pi_{D_1}(t) \in T_1 \wedge \pi_{D_2}(t) \in T_2$. Here $\pi_C(t)$ is the projection of trace t over dependence D onto dependence $C \subseteq D$, which not only can remove some symbols from the trace but also make some of them independent. Consider for example trace $t = [abcd]_{D_N} = \{abcd, acbd\}$ where N is the net from Figure 1 and D_N is its dependence. Take $D_1 = D_N \setminus \{a, c\}^2$ and $D_2 = \{b, d\}^2 \cup \{c, d\}^2$, then $\pi_{D_1}(t) = \{abcd, acbd, cabd\}$ and $\pi_{D_2}(t) = \{bcd, cbd\}$.

We present now a constructive method for computing synchronization of trace languages. It can be used in the algorithm in Listing 1. Later on we present how to adapt the method from net languages to trace languages and discuss how it can be implemented in an efficient way with dependence graphs.

Let us start with the observation that synchronization of singleton trace languages yields either a language consisting of one trace, being either non-empty or empty (epsilon) sequence. Therefore, for traces t_1 over D_1 and t_2 over D_2, we define $t_1 \parallel t_2$ to be t iff $\{t_1\} \parallel \{t_2\} = \{t\}$. Recall that $t_1 \parallel t_2$ is a trace over $D_1 \cup D_2$. Such definition allows one to compute the synchronization of trace languages as the set of all synchronizations of the pairs of the individual traces

in the Cartesian product of those two trace languages: $T_1 \parallel T_2 = \{t_1 \parallel t_2 \mid t_1 \in T_1, t_2 \in T_2\}$.

We can observe that the result of this operation always contains empty (epsilon) sequence and contains any other sequences only if some traces match on the common *interface* $Int = \Sigma(D_1) \cap \Sigma(D_2)$ [1]. Formally, this requirement means that for $[seq_1]_{D_1} = t_1$ and $[seq_2]_{D_2} = t_2$, we have $\pi_{Int}(seq_1) = \pi_{Int}(seq_2)$. This follows immediately from the definition of synchronization.

The algorithm to construct a representative for the output trace from matching representatives of the synchronized traces would be to divide the synchronized representatives into subsequences delimited by interface symbols and then interweave in any way the subsequences from the corresponding positions. Let $t_1 = [k_{(*,1)}, i_1, k_{(1,2)}, i_2, k_{(2,3)}, \ldots, i_n, k_{(n,*)}]$ and $t_2 = [l_{(*,1)}, i_1, l_{(1,2)}, i_2, l_{(2,3)}, \ldots, i_n, l_{(n,*)}]$, where $k_{(a,b)}$ and $l_{(c,d)}$ are some sequences and $i_1, i_2, \ldots, i_n$ some symbols and Int the common interface such that $i_1, i_2, \ldots i_n \in Int$ and none of $k_{(a,b)}$ and $l_{(c,d)}$ contains any letters from Int. At first it may seem that $t_1 || t_2 = [k_{(*,1)}, l_{(*,1)}, i_1, k_{(1,2)}, l_{(1,2)}, i_2, k_{(2,3)}, l_{(2,3)}, \ldots, i_{n-1}, k_{(n-1,n)}, l_{(n-1,n)}, i_n, k_{(n,*)}, l_{(n,*)}]$, which is obtained by reversing the definition of projection and the observation that $k_{(x,x+1)}$ and $l_{(x,x+1)}$ cannot share any symbols, as they would have to be in Int (so the interweaving of their symbols is arbitrary as long they are between symbols i_x and i_{x+1}).

Unfortunately, it may happen that for two traces, some representatives allow for such a match but not all. For example traces $t_1 = [abc]_{D_1}$ over $D_1 = \{a, b\}^2 \cup \{b, c\}^2$ and $t_2 = [cb]_{D_2}$ over $D_2 = \{(b, b), (c, c)\}$ and common alphabet $\{b, c\}$ can be synchronized. This becomes apparent if bc is chosen as the representative for t_2. Next we present a constructive method that does not require us to enumerate all possible representatives to check if traces can be synchronized or not.

7 Dependence graphs and prefix closeness property

In this section we describe how synchronization of two singleton trace languages $\{t_1\}$ and $\{t_2\}$ can be implemented in an efficient way. For that we extend the *dependence graphs* representation from [3]. We also observe that the languages we deal with are prefix closed, which allows us to further minimize the amount of memory needed to store them. The constructive synchronization we present, is designed in such a way that it results in the synchronization of the longest possible prefixes, without storing them independently.

Dependence graphs are an alternative method to represent traces. They combine the partial ordering of transition occurrences common to all sequences in the dependence relation, in a form of a graph of partial ordering relation.

Formally, a dependence graph is a directed, acyclic graph that is labelled by elements of the alphabet (symbols from T). Only nodes that represent dependent transitions are connected by an edge. Example of dependence graphs are given in the left hand side of Figure 2. They represent exactly the same information

[1] Note that it is possible for a symbol to be independent of any other symbols.

as $[adbad]_{D_1} \parallel [adcead]_{D_2} = [adbcead]_D$, where $D_1 = \{a, b, d\}^2$, $D_2 = \{a, c, e\}^2 \cup \{c, d\}^2$ and $D = D_1 \cup D_2$.

The intuition is that the synchronization operation corresponds to combining the dependence graphs by merging the interface nodes. That is, there has to be the same number of occurrences of every interface transition, and after merging the interface transitions together we have to get a dependence graph — which is directed and acyclic, i.e., the ordering defined by the synchronized dependence graphs cannot be contradictory.

As dependence relation is reflexive, we extend the notation of Mazurkiewicz and enumerate successive occurrences of every transition in the trace/dependence graph. With such an indexing the occurrences of transitions are unique and the dependence graph representation is common to all representatives of a trace. We can now observe, that if dependence graphs have compatible indexing and ordering on the common interface, i.e., the order of occurrences is not contradictory and there is exactly the same number of each of the common interface transitions in both dependence graphs, then we can synchronize them by merging nodes representing the common interface (see Figure 2).

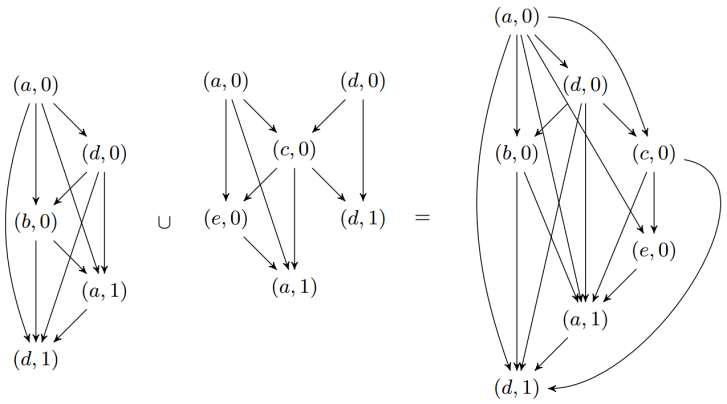

Fig. 2: Synchronization of two dependence graphs calculated by merging of their indexed representations.

Now is the moment to observe that net languages and their trace representations are *prefix closed*, i.e., for every sequence seq or trace $[seq]$, all their prefixes are also in the language, i.e., are sequences/traces of the same system. Thus we do not have to remember the prefixes, which even further compresses our trace representation. Cleaning a huge language of prefixes would be computationally intensive, especially as we do not have linear ordering of events for traces. Therefore we developed a synchronization algorithm that not only synchronizes traces (in the form of dependence graphs), but if they are not synchronizable then prefix synchronizes them, i.e., it finds the synchronization of their longest syn-

chronizable prefixes. This allows us to avoid storing most of the prefixes of other existing traces, while reducing the size of the representation even further. Some prefixes can still appear in the representation, because it is possible that the result of prefix synchronization of some traces is a prefix of some other trace. We leave the elimination of this redundancy for future research.

For dependencies $D_1 = \{a, b, c, d\}^2$ and $D_2 = \{a, b, e\}^2$ with a common interface being comprised of a, b, and for traces $t_1 = [aacdba]_{D_1}$ and $t_2 = [aaeaa]_{D_2}$, we have $t_1 \parallel t_2 = \epsilon$. If we consider prefixes $t'_1 = [aacd]_{D_1}$ and $t'_2 = [aae]_{D_2}$, we can synchronize them and get $t'_1 \parallel t'_2 = [aacde]_D = \{aacde, aaced, aaecd\}$.

The following steps can be used to extend the algorithm from Listing 1 to a trace version with prefix synchronization.

1. Compute the common interface (this can be done once for two trace languages and reused for all the synchronized trace pairs).
2. Check if the number of occurrences of every transition from common interface is equal in both traces — interfaces match. If not, for each element of the interface take the minimum of the numbers of occurrences of that element in both traces and remove the occurrences of this transition indexed above this minimum. Remove also all the successors of removed transitions. Repeat if signatures still do not match.
3. Merge the indexed dependence graphs. If there are any cycles, remove all their nodes. Remove also all descendant nodes of the removed ones.
4. The resulting dependence graph is the result of synchronization of longest synchronizable prefixes.
5. From the dependence graph representation pick some representative for that trace. Use the same method of choosing representatives for all dependence graphs, i.e., make the representatives canonical. If the dependence graph has more nodes than the limit k pick all the prefixes of length k and choose representatives for them.

In step 2 we find the prefixes that are suitable for synchronization by merging of dependence graphs representations. We remove nodes that cause the incompatibility of common interfaces, because the dependence graphs or traces that do not have matching interfaces are not synchronizable. We also remove all successive transitions.

A simple example shows that such a removal of nodes may have to be repeated several times. Consider $D_1 = \{a, b, c\}^2 \cup \{a, d\}^2$, $D_2 = \{a, b, c\}^2$ with the common interface $\{a, b, c\}$ and traces $t_1 = [adcb]_{D_1}$ and $t_2 = [abbc]_{D_2}$. We have to make three steps (remove b then c then d) until we find prefixes that have matching interfaces.

These steps are presented in Figure 3. The left hand side dependence graphs are the subsequently examined prefixes of t_1, and on the right hand side are the prefixes of t_2. In the first step, there are more occurrences of b in t_2, so we remove the second one, and also remove the only node labelled c. This causes the situation, in which there is one more occurrence of c in the left hand side dependence graph. To remove it we also need to remove the only node with b.

Again there is no match on the common interface $\{a, b, c\}$, so in the third step we have to remove the node with b from right hand side dependence graph. This results in prefixes equivalent to traces: $[ad]_{D_1} \sqsubseteq t_1$ and $[a]_{D_2} \sqsubseteq t_2$, which can be synchronized.

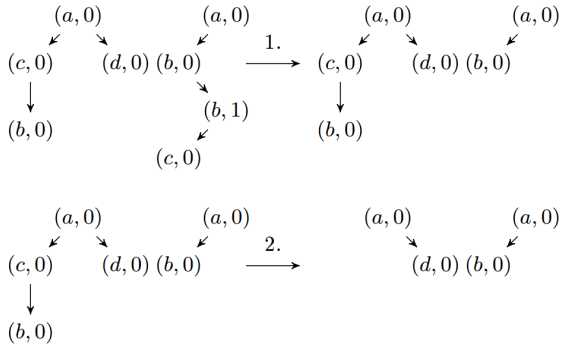

Fig. 3: Finding longest prefixes with matching interfaces.

Note that even if two traces have matching interfaces, it does not necessarily mean it is possible to synchronize them. It may be the case that their common interface from some point shows different orderings of transition occurrences, and the merging of dependence graphs will create cycles. Thus in step 3 we look for cycles. All cycle nodes can be eliminated from the result (with all of their successor nodes), because this corresponds to taking prefixes of source operands. Note that we cannot hope to break a cycle by taking a prefix in just one trace as this would prevent matching on the common interface.

Removing the same nodes from the source dependence graphs, which have been deleted during cycle removal, maintains common interface compatibility.

During synchronization only transitions that are included in the common interface are merged together, so the size of the result of synchronization can be larger than of any of the components. As we are interested in traces of limited length, we may calculate prefixes of limited length. Note that trace can have more then one prefix of any given length. For example, consider dependence relation $D = \{a, b\}^2 \cup \{a, c\}$ and trace $t = [abc]_D$. It has two prefixes of length 2: $[ab]_D$ and $[ac]_D$. This operation can be easily implemented for dependence graphs where we can iteratively drop nodes with no outgoing edges. We do not lose any information this way, because all traces of length bounded by k can be obtained by synchronization of traces of length bounded by k. We found this optimization to be profitable in our experiments, but it does not have to be so in general, as it may be better to store longer traces rather than many prefixes.

When choosing a representative for the trace, we make sure that for the same graph we always choose the same representative, so that if two synchronizations have the same result, we can eliminate one of them as a duplicate.

8 Experimental evaluation

We conducted tests of the experimental implementation. The Petri net examples being simulated were adapted from the papers of Mazurkiewicz, together with their initial markings. Below we use the name m95 for the sole example from [3], and m84_Y where Y is a number, to indicate the example number Y from [2]. In order to get even larger nets, we multiplied them, by creating a few disjoint copies of the initial net, adding a starting place and a starting transition, which consumes the token from the starting place and inserts tokens into the places which have been originally marked in each of the copies. Figure 4 presents an example of this construction, where net m95 is three-fold multiplied to yield m95x3. Note that the disjoint copies of the initial m95 net are independent in m95x3.

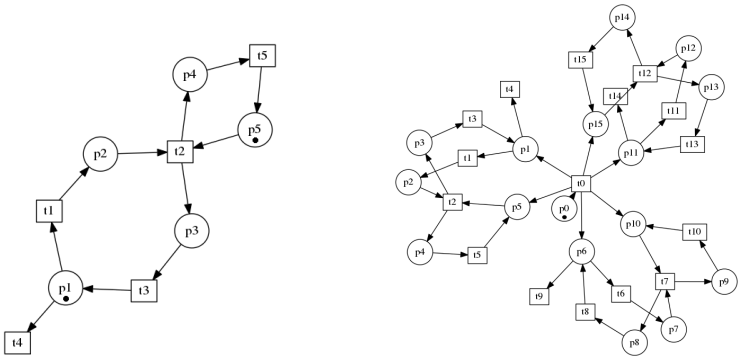

Fig. 4: Petri net multiplication: m95 (on the left) and m95x3 (on the right).

The test cluster consisted of 12 machines equipped with Intel Core Quad Q9550 2,8GHz processors with 4 cores and 4 GB RAM each. Two of the machines were masters: one for Spark and one for HDFS. The other machines were used as Spark workers for distributed computation and data nodes for HDFS for the purpose of saving the final results.

The nodes were running Java Development Kit 1.80_60, Apache Spark framework 1.4.1, and HDFS from the Hadoop 2.7.1 distribution. The Java VM was restricted to use at most 2 GB of memory.

Initially we performed several tests to choose the right number of partitions. It turned out, that in general a good choice was to use the number of partitions equal to the number of workers, i.e., 10, and enforce repartition after each task

to equalize their load. However, this choice was not absolutely superior, and in some cases increasing the number of partitions to four times the number of workers, i.e., 40, was beneficial.

Then we measured running time as a function of the number of places and transitions in the input net. The results are depicted in Figure 5. The general observation is that the graphs are almost linear in the logarithmic scale, so indeed the running time increases exponentially with the size of the net, if its structural complexity does not grow — which in our case comes from the fact that each of the lines represents a sequence of nets, created by increasing the number of independent copies of the initial one.

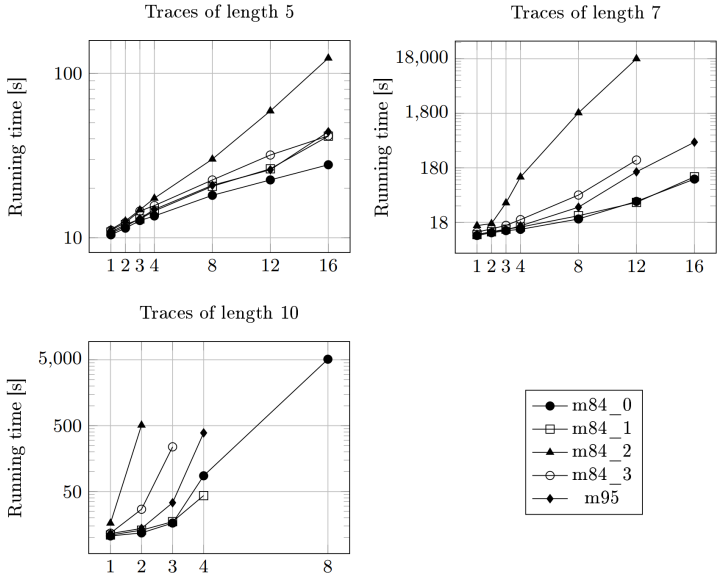

Fig. 5: Running times of the PetriSpark application. The horizontal axis indicates how many times the initial net has been multiplied. So 1 corresponds to the initial net m, then the consecutive numbers stand for mx2, mx3, mx4, mx4x2, mx4x3 and mx4x4, having about 2, 3, 4, 8, 12 and 16 times more places and transitions than m.

Concerning the memory usage (see Figure 6), we have chosen to count data elements stored in RDDs, rather than the total memory consumption. The benefit is that this gives a hint about the true sizes of the sets that emerge during computation, without taking their implementation details into account.

Those sizes vary greatly, depending on point of each step when they were measured, and to show this we represent the logs as a few separate graphs.

The curve indicated as *initial* represents the size of data structures at the beginning of the step. The curve described as *after synchronization* provides the size of data structures after the traces are synchronized, yet those traces can still be longer than the requested limit. Next, we calculate prefixes of limited length which can increase the data size and is presented as curve described as *after prefixes*. Finally, duplicates are removed by executing *distinct*, which eliminates the obvious redundant data elements and produces the initial data for the next step.

The peaks on the graphs generally occur when many duplicates emerge. This is caused by the structural properties of the nets being simulated.

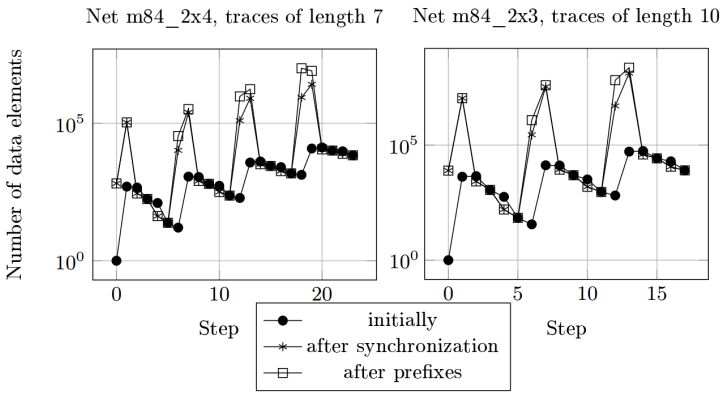

Fig. 6: The number of data elements during execution.

Finally, we present a graph, depicting how the optimizations we perform (elimination of prefixes and replacing sequences with traces) influence the size of the resulting sets of objects, which represent the complete set of runs of the system. We have the following four cases: (A) is the unoptimized case, (B) is the case of all sequences of length up to k (with proper prefixes eliminated), (C) is the case of all traces of length up to k (stored by using only one representative) and (D) of all traces of length k, without proper prefixes, i.e., with both optimizations used simultaneously.

In order to visualize the situation, we repeated the experiments for all four possible cases (A), (B), (C) and (D) for the following 15 nets [2]: m84_0, m84_0x2, m84_0x3, m84_1, m84_1x2, m84_1x3, m84_2, m84_2x2, m84_2x3, m84_3, m84_3x2, m84_3x3, m95, m95x2, m95x3, and for the length limits $k = 5,\ 7$ and 10.

Since we had many nets of different sizes, we normalized the results by calculating, for each net and each k separately, the ratios of (D), (C), (B) and (A) to

[2] Recall that m84_2x3 is the second example from [2] multiplied three times.

(A), getting the ratio of size reduction in each case. Then we computed averages of the ratios over all nets. As it is now apparent from the graph in Figure 7, representing behaviours in the form of traces reduces the sizes of the datasets much more than eliminating prefixes. It is also visible that the larger k we have, the better the effect of moving from sequences to traces. Eliminating prefixes alone gives a constant rate of size reduction, independent of k.

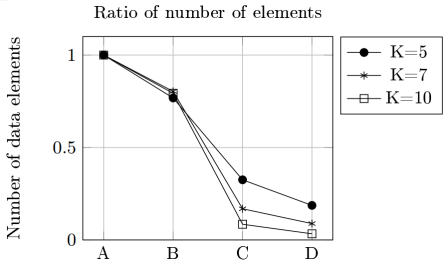

Fig. 7: The average representation size reduction over 15 nets, for $k = 5,\ 7,\ 10$. (A) no optimization, (B) eliminating prefixes, (C) moving to traces, (D) moving to traces and prefix elimination.

9 Summary and future research

Our research demonstrates a successful attempt to treat the investigation of the runs of a PN as a Big Data problem. Our contributions are the following: (1) Distributed generation of all runs, and moreover by structural induction over the net being simulated; (2) Two optimizations, which reduce the size of the resulting dataset manifold. Our tool can be applied for conformance checking, where a PN model of a process is compared to its real-life behaviour, recorded as the set of logs of its runs.

Our approach permits further improvements. An extra step could be added to the algorithm to remove all prefixes which would require developing specialized algorithm for that and testing if the extra effort is profitable. Note that for traces which do not represent linear ordering of events this may not be as straightforward as for sequences. In the case that the simulated net contains several copies of a certain subnet, we plan to reuse the trace language computed for one copy while processing other copies, instead of recomputing it from scratch. We intend to automatize detection of opportunities for such reuse.

References

1. IEEE Task Force on Process Mining. Process Mining Manifesto. In F. Daniel, K. Barkaoui, and S. Dustdar, editors, *Business Process Management Workshops*,

volume 99 of *Lecture Notes in Business Information Processing*, pages 169–194. Springer-Verlag, 2012.
2. A. W. Mazurkiewicz. Semantics of concurrent systems: a modular fixed-point trace approach. In *Advances in Petri Nets 1984, European Workshop on Applications and Theory in Petri Nets, covers the last two years which include the workshop 1983 in Toulouse and the workshop 1984 in Aarhus, selected papers*, pages 353–375, 1984.
3. A. W. Mazurkiewicz. Introduction to trace theory. In V. Diekert and G. Rozenberg, editors, *The Book of Traces*, pages 3–41. World Scientific, 1995.
4. W. Reisig. *Petri Nets: An Introduction*. Springer-Verlag, 1985.
5. G. Rozenberg. Behaviour of elementary net systems. In *Advances in Petri Nets 1986, Part I on Petri Nets: Central Models and Their Properties*, pages 60–94, London, UK, UK, 1987. Springer-Verlag.
6. W. M. P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, 1st edition, 2011.
7. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.