Control and Cybernetics

vol. **42** (2013) No. 3

Visualisation of concurrent processes*

by

Łukasz Mikulski and Marcin Piątkowski

Faculty of Mathematics and Computer Science
Nicolaus Copernicus University
87-100 Toruń, ul. Chopina 12/18
{lukasz.mikulski, marcin.piatkowski}@mat.umk.pl

Abstract: Mazurkiewicz traces are a widely used model for describing the languages of concurrent systems computations. The causal structure of atomic actions occurring in a process modeled as a trace generates a partial order. Hasse diagrams of such order are very common structures used for presentation and investigation in the concurrency theory, especially from the behavioural perspective. We present effective algorithms for Hasse diagrams construction and transformation. Later on, we use them for enumeration of all linearisations of the partial order that represents a concurrent process. Additionally, we attach the flexible visual implementation of all considered algorithms.

Keywords: concurrency, partial order, Hasse diagram, directed acyclic graph, linearisation, Mazurkiewicz traces

Introduction

Concurrent systems are extensively investigated through the perspective of the causal structure of their actions (understood as the most detailed elements of the system behaviour), which leads to the respective formal language semantics. An example of utilizing causal structures is IBM analytic tool InfoSphere (see IBM, 2008) that operates on job sequences, see also Stevenson (2007). On the other hand, in many theoretical models such causal structure of atomic actions establishes the partial order. Very good examples are Petri nets (see Petri, 1962) and Mazurkiewicz traces (see Cartier and Foata, 1969; Mazurkiewicz, 1977), where a trace is understood as the class of equivalent sequential computations of the considered Petri net. Graphs of traces (implied by the dependency relation), as well as their transitive closures (graphs of considered partial orders) are usually illegible. Therefore, the Hasse diagrams, which are transitive reductions of partial orders, are widely adopted for the graphical visualisation in the poset and trace theories, see Diekert and Rozenberg (1995), Gastin (1990), Graham et al.

*Submitted: June 2013; Accepted: September 2013.

(1995). Although a single trace τ combines equivalent (from the behavioral point of view) sequences of actions, its Hasse diagram is unique and unambiguous. Moreover, it is worthwhile to notice that every partially ordered set can be defined by a trace with the same Hasse diagram, see Mikulski, Piątkowski and Smyczyński, (2011).

The proper visualisation of graphs is a very popular topic. In the case of Hasse diagrams, there are some studies contributing to this issue, see, in particular, Freese (2004a), Garg and Tamassia (1994). Some other related works are discussed in section 5.1. There are also some papers concerning the problem of maintaining the transitive reduction of a directed acyclic graph. See, for instance, Poutré and van Leeuwen (1987), where the operations of insertion and deletion of single edges are investigated. From the computational complexity point of view, the strict relationship between the transitive reduction and the transitive closure were deeply studied (see Aho, Garey and Ullman, 1972). In the general case, both problems may be solved using boolean matrix multiplication. However, in practice, some less efficient methods (with respect to the time complexity) are used (see Eve and Kurki-Suonio,1977; Poutreánd van Leeuwen, 1987; Purdom, 1970). As far as we know, none of existing approaches utilizes the relationship between the causal structure of traces and partially ordered sets. Furthermore, editing the visualised poset structure in available software tools is difficult or impossible. To fill this gap we provide an application DiaDem,by Mikulski and Piątkowski (2012), which implements algorithms described in this paper and allows for the investigation of this topic from the point of view of the trace theory.

This paper describes several algorithms that not only construct the Hasse diagram but also transform its structure. Finally, these procedures are used to generate all representatives of a trace (the linearisations of its causal order). The concept of the provided algorithms is based on a notion of the *tail* of the trace – the special data structure enriching the diagram. The approach presented here is the extension of the results of Mikulski, Piątkowski and Smyczyński (2011), where the tail structure was first defined.

We start with introducing some basic definitions and notations. Then we investigate the needed data structures and present simple algorithm for the Hasse diagram construction. Further, we extend this simple approach to achieve the possibility of efficient diagram reconstruction (by removing maximal elements). The next section is devoted to the enumeration of all representatives of a trace. Finally, we present a short survey of existing software related to this topic, including the application DiaDem. The paper is concluded by pointing out some directions for future work. A few examples, which illustrate the running details of the presented algorithms are shown in the appendix Examples.

1. Basic notions

1.1. Algebra of actions

Throughout the paper we use the standard notions of the formal language theory, see Hopcroft and Ullman (1979), Rozenberg and Salomaa (1997). In particular, by an *alphabet* we mean a nonempty finite set Σ, the elements of which are called *(atomic) actions*. Finite sequences over Σ are called *words*. The set of all finite words over Σ, including the empty word λ, is denoted by Σ^*. We assume that alphabet Σ is given together with a total order $\leq$, called *lexicographical order*, and extend it in a natural way to the level of words.

Let $w = a_1 \dots a_n$ and $v = b_1 \dots b_m$ be two words. Then the concatenation of w and v is defined as $w \circ v = wv = a_1 \dots a_n b_1 \dots b_m$. The alphabet $alph(w)$ of w is the set of all actions occurring within w, and $\#_a(w)$ is the number of occurrences of an action a within w. The set $occ(w)$ of *action occurrences* of w comprises all pairs (a, i) such that $a \in alph(w)$ and $1 \leq i \leq \#_a(w)$. The *head* (first action occurrences) and the *tail* (last action occurrences) of a word w are two sets defined by:

$$head(w) = \{(a, 1) \mid a \in alph(w)\} \quad \text{and} \quad tail(w) = \{(a, \#_a(w)) \mid a \in alph(w)\} .$$

Let $\alpha = (a, i)$ be an action occurrence in $occ(w)$. The position $pos_w(\alpha)$ of α within w is the smallest integer j such that $\#_a(a_1 \dots a_j) = i$, and $\ell(\alpha) = a$ is the *label* of α (the projection onto the first coordinate). We can apply ℓ to sequences and sets of action occurrences in the usual way, i.e:

$$\ell(\alpha_1 \dots \alpha_n) = \ell(\alpha_1) \dots \ell(\alpha_n) \quad \text{and} \quad \ell(\{\alpha_1, \dots, \alpha_n\}) = \{\ell(\alpha_1), \dots, \ell(\alpha_n)\}.$$

1.2. Posets

The composition of two binary relations, R and Q, over a set X is given by $R \circ Q = \{(a, b) | \exists_{x \in X} : aRx \wedge xQb\}$. Given a relation $R \subseteq X \times X$, R^0 is the identity relation on X, and $R^n = R^{n-1} \circ R$, for all $n \geq 1$. The *transitive closure* of a relation R is $R^+ = \bigcup_{i \geq 1} R^i$.

A *directed acyclic graph* is a pair $dag = (X, R)$, where X is a finite set and R is acyclic irreflexive binary relation on X. In a diagrammatical representation, X is the set of vertices, while R is the set of arcs of dag. If R is transitive, a pair (X, R) is called a *poset* and usually denoted by $po = (X, \prec)$. Moreover, every directed acyclic graph $dag = (X, R)$ such that $(X, R^+) = po$ is called *po-diagram*. Among all the *po*-diagrams, we can distinguish the smallest one (i.e. the one with the smallest relation R), denoted by $H(po) = (X, \prec^{\mathsf{cov}})$, and called the *Hasse diagram* of po. Note that $\prec^{\mathsf{cov}}$ can be obtained from $\prec$ by simply removing all the arcs implied by the transitivity of $\prec$; in other words, $\prec^{\mathsf{cov}} = \prec \setminus \prec \circ \prec$. Moreover, if (X, R) is a *po*-diagram, then $\prec^{\mathsf{cov}} = R \setminus \bigcup_{i \geq 2} R^i$.

A *linearisation* (also called a *linear extension*) of a poset $po = (X, \prec)$ is a sequence $u = x_1 \dots x_n$ of distinct elements of po such that $X = \{x_1, \dots, x_n\}$

and, for all $1 \leq i < j \leq n$, $x_j \not\prec x_i$. Observe that in every linearisation the first element x_1 is one of the minimal elements of $(X, \prec)$.

A *chain* of a poset $po = (X, \prec)$ is a sequence $x_1 \dots x_m$ of distinct elements of po such that $C = \{x_1, \dots, x_m\} \subseteq X$ and, for all $1 \leq i < j \leq m$, $x_i \prec x_j$. We say that a chain is maximal if there is no element $y \in X$ such that for every $1 \leq i \leq m$ $x_i \prec y$ or $y \prec x_i$. Note that in partial order theory a chain is usually defined as a set $\{x_1, \dots, x_m\}$ with above the properties, not as a sequence. However, such a sequence is unique and makes the definition closer to the definition of the linearisation.

Both linearisation and maximal chain approximate a partial order by a total order. The very important and useful fact about the sets of all such objects is that they constitute initial poset uniquely. In the case of linearisations it is known as Szpilrajn theorem (Szpilrajn, 1930), while in the case of maximal chains it is a simple corollary to the Dilworth's decomposition theorem for partially ordered sets (Dilworth,1950).

1.3. Traces (equivalence classes)

A *concurrent alphabet* is a pair $\Psi = (\Sigma, dep)$, where Σ is an ordered alphabet and $dep \subseteq \Sigma \times \Sigma$ is a reflexive and symmetric *dependence* relation. The corresponding *independence* relation is given by $ind = (\Sigma \times \Sigma) \setminus dep$.

A concurrent alphabet Ψ defines an equivalence relation $\equiv_\Psi$ identifying words which differ only by the ordering of independent actions. Two words $w, v \in \Sigma^*$ satisfy $w \equiv_\Psi v$ if there exists a finite sequence of commutations of adjacent independent actions transforming w into v. More precisely, $\equiv_\Psi$ is a binary relation over Σ^*, which is the reflexive and transitive closure of the relation $\sim_\Psi$, such that $w \sim_\Psi v$ if there exist $u, z \in \Sigma^*$ and $(a, b) \in ind$ satisfying $w = uabz$ and $v = ubaz$.

Equivalence classes of $\equiv_\Psi$ are called *Mazurkiewicz traces* (see Diekert and Rozenberg, 1995; Mazurkiewicz, 1977; Mikulski, 2008), and the trace containing a given word w (called a *representative* of the trace) is denoted by $[w]$. The set of all traces over Ψ is denoted by $\Sigma^*/_{\equiv_\Psi}$, and the pair $(\Sigma^*/_{\equiv_\Psi}, \circ)$ is a (trace) monoid, where $\tau \circ \tau' = [w \circ w']$, for any words $w \in \tau$ and $w' \in \tau'$, is the concatenation operation for traces. Note that trace concatenation is well-defined as $[w \circ w'] = [v \circ v']$, for all $w, v \in \tau$ and $w', v' \in \tau'$. Similarly, for every trace $\tau = [w]$ and every action $a \in \Sigma$, we can define:

$$alph(\tau) = alph(w) \qquad occ(\tau) = occ(w) \qquad \#_a(\tau) = \#_a(w)$$
$$head(\tau) = head(w) \qquad tail(\tau) = tail(w).$$

2. Poset of a trace

One can represent a trace τ as a poset of action occurrences. More precisely, $po(\tau) = (occ(w), \prec_w^+)$ is the poset *induced* by τ, where w is any word belonging to τ, and $\prec_w$ is a binary relation on $occ(w)$ such that $\alpha \prec_w \beta$ if $pos_w(\alpha) < pos_w(\beta)$ and $(\ell(\alpha), \ell(\beta)) \in dep$. Transitive reductions of $(occ(w), \prec_w^+)$ and

$(occ(w), \prec_w)$ are equal and form the Hasse diagram of the trace τ (namely $(occ(w), \prec_w^{cov})$), hence $\prec_w^{cov} \subseteq \prec_w \subseteq \prec_w^+$. Moreover, the linearisations of $(occ(w), \prec_w^{cov})$ correspond to the representatives of a trace $[w]$. Together with an order $\leq$ on actions this allows for defining the lexicographically minimal and maximal representatives of the trace $[w]$ (hence minimal and maximal linearisations of $(occ(w), \prec_w^{cov})$). Such representatives/linearisations are called the normal forms of a trace, and denoted by *minlex* and *maxlex*, respectively.

We define the *border* of a trace τ, denoted by $bord(\tau)$, as the subset of its tail consisting of the maximal elements of the poset $(occ(w), \prec_w^+)$. Note that the notions of head, tail and border of a trace τ could be considered also for its Hasse diagram $\mathcal{G}(\tau)$. In this case, the border of $\mathcal{G}(\tau)$ is the set of nodes without outgoing arcs, while the head and the tail of $\mathcal{G}(\tau)$ are the sets of nodes never preceded or followed, respectively, by a node with the same label. Precisely, for all paths containing a node V from the head (the tail) of $\mathcal{G}(\tau)$ all predecessors (successors) of V have labels different than $\ell(V)$. Note that occurrences of actions have both language theory and graph characterization. Note, further, that we use the initial Greek letters (α, β, γ) to denote the occurrences of actions in a trace τ, and V (possibly with indexes) – the occurrences of actions in Hasse diagram of τ.

For every occurrence α we define the special subset of its predecessors (elements β in relation $\prec_w^+$ with α) called the *sources* of α. Namely, from every nonempty set of all predecessors of α with a given label b we choose one representative $\beta = (b, i)$. The chosen occurrence is the one with the greatest second coordinate, i.e. the rightmost (preceding α) occurrence of b. In other words, β is a source of α if $\beta \prec_w^+ \alpha$ and $(\gamma \prec_w^+ \alpha \wedge \gamma \neq \beta \wedge \ell(\gamma) = \ell(\beta)) \Rightarrow \gamma \prec_w^+ \beta$ (see Example 1). Note that α is never its own source.

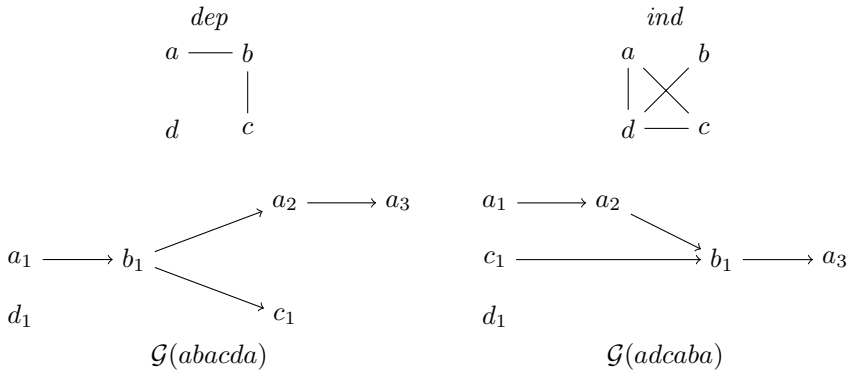


Figure 1. The graphs of examples dependence and independence relations and Hasse diagrams of the words $w_1 = abacda$ and $w_2 = adcaba$

Example 1 *We equip the alphabet* $\Sigma = \{a, b, c, d\}$ *with dependence and independence relations dep and ind. The graphs of these relations and the Hasse*

diagram of the words $w_1 = abacda$ *and* $w_2 = (w_1)^R = adcaba$ *are depicted in Fig. 1. In this case the words* $w_1 = ab\mathbf{acda}$ *and* $w_3 = ab\mathbf{caad}$ *are equivalent. The tail of* $\mathcal{G}(abacda)$ *is* $\{a_3, b_1, c_1, d_1\}$. *The sources of the node* c_1 *in this diagram are* a_1 *and* b_1. *Note that* a_1 *is the source of* c_1 *while actions* a *and* c *are independent, so* a_1 *and* c_1 *are not in relation* $\prec_w$. *However, because of the existence of the occurrence* b_1 *which has to appear after* a_1 *and before* c_1 *we have* $a_1 \prec_w^+ c_1$. *Moreover, it is worth noting that the second occurrence of* a *in the generating sequence (i.e.* $abacda$*) appears before the first occurrence of* c*, but it is not a source of* c_1.

From the diagrammatical point of view, the node V_1 is a source of the node V_0 if path from V_1 to V_0 includes a node (other than V and V_0) with label equal to $\ell(V_1)$. More formally:

Proposition 2.1 *Let* w *be a word over a concurrent alphabet* (Σ, dep) *and* $\mathcal{G}(w)$ *be its Hasse diagram. The node* V_1 *is a source of the node* V_0 *if and only if all paths in* $\mathcal{G}$ *from* V_1 *to* V_0 *do not include nodes (distinct from* V_1 *and* V_0*) labeled with* $\ell(V_1)$.

Proof. At the beginning, let us notice that for all vertices $V,\ V'$, if $\ell(V) = \ell(V')$ then $V \prec_w^+ V'$ or $V' \prec_w^+ V$. Hence, there exists a path between V and V'. Let us consider the set $S(V_0, a)$ of all vertices with label a that precedes V_0. Formally, $S(V_0, a) = \{V; V \prec_w^+ V_0 \ \wedge\ \ell(V) = a\}$. Suppose that the set $S(V_0, a)$ is not empty. Otherwise V_0 do not have a source labeled with a. Let V_{max} be the greatest element in $S(V_0, a)$. Since all nodes from $S(V_0, a)$ have the same label, such an element exists. Moreover, for every other element $V \in S(V_0, a)$ we have $V \prec_w^+ V_{max} \prec_w^+ V_0$ and there is no element of label a which is greater than V_{max} and smaller than V_0. This means that V_{max} is the only vertex labeled with a that satisfies the left hand side of equivalence stated in the proposition. Moreover, V_{max} is the only vertex labeled with a that satisfies the right hand side of those equivalence, which completes the proof. ■

3. Hasse diagram data structure

In this section, we introduce a data structure that is useful in operations on the Hasse diagram of partially ordered set generated by the trace $\tau = [w]$. The structure consists not only of the graph representing Hasse diagram $H(po(\tau))$, but also some auxiliary lists and sets, which are used in the proposed procedures of building and rebuilding Hasse diagrams.

The main extension of the structure of $H(po(\tau))$ is the list that consists of the references to all nodes contained in $tail(w)$. The order of the elements in the *tail list*, denoted by RT, is the inverted order of pos_w of considered occurrences. Thereby, the first element of the tail list is the occurrence of the last action contained in w.

We enhance every node of $\mathcal{G} = H(po(\tau))$ by the set of all its existing sources. In fact, we divide the set of all existing sources of V into two sets – the set of

sources, which belongs to the tail, denoted by *src*, and other sources, denoted by *del_src*. It is worth noticing that the *del_src* part of the sources set is necessary only when we modify an existing diagram, i.e. during the recovery of the tail structure after removing a node from the border. To create a new node and append it to the existing diagram we need the sets of sources which belong to the current tail only. We make use of this remark in the procedure called SIMPLE-APPEND. For technical reasons, we also add the integer *id* consistent with the order of pos_w of the occurrences. If we never use the REMOVE procedure, then *id* for an occurrence α is equal to $pos_w(\alpha)$.

3.1. Online algorithm for building a Hasse diagram

This section describes the construction of the Hasse diagram. We present an online algorithm, which for a given word w over a concurrent alphabet (Σ, dep), computes the Hasse diagram of the trace $\tau = [w]$. To denote the size of a trace (number of occurrences of single actions) we use the letter n, while the letter k denotes the size of the alphabet Σ.

In the SIMPLE-APPEND procedure, which is used to build the diagram $\mathcal{G}$ only, we do not take care of the sets *del_src*. We also update the sets *src* for the elements of $tail(\mathcal{G})$ only. We do not clear the sets of sources for the nodes which are removed from $tail(\mathcal{G})$. Note that, due to Proposition 3.1, to be formulated in this Section, all arcs adjacent to a node V are present in $\mathcal{G}$ before removing V from $tail(\mathcal{G})$. Let us first introduce the pseudocode of the procedure that builds a Hasse diagram from an arbitrary representative of a trace.

Algorithm 1: Hasse diagram building

Input: a word $w = w_1 w_2 \cdots w_n$ over a concurrent alphabet (Σ, dep)
Output: a graph $\mathcal{G}$ representing a Hasse diagram

```
1 G := empty graph;
2 RT := ∅;
3 for i := 1 to n do
4 |  SIMPLE-APPEND(G,RT,w_i);
```

Algorithm 1 is an example of an online algorithm. Its correctness follows from the correctness of a single step extracted as the procedure SIMPLE-APPEND. Since the procedure SIMPLE-APPEND has the time complexity of $O(k^2)$ and we call it for every occurrence from linearisation, the time complexity of the Algorithm 1 is $O(nk^2)$.

Operation SIMPLE-APPEND

The procedure SIMPLE-APPEND operates on the graph $\mathcal{G}$, equipped with an additional structure of the tail, denoted by RT. It appends a new node, labeled with an action passed as an argument, to the graph $\mathcal{G}$. The most important

fact that stands for this algorithm, describes the role of the tail in appending a single occurrence to a Hasse diagram, and is formulated in Proposition 3.1.

Lemma 1 *Let w be a word over a concurrent alphabet (Σ, dep) and let the graph $\mathcal{G} = H(po([w]))$ be its Hasse diagram. If in the Hasse diagram $\mathcal{G}$ there exists an arc from V_1 to V_0 then V_1 is a source of V_0.*

Proof. Recall that $\mathcal{G}$ is the graph of $(occ(w), \prec_w^{cov})$. Directly by the definition of the transitive reduction, if there is an arc (V_1, V_0) in $\mathcal{G}$, then this arc is the only path from V_1 to V_0. Hence V_1 is a source of V_0. ■

Proposition 3.1 *Let w be a word over a concurrent alphabet (Σ, dep) and let the graph $\mathcal{G} = H(po([w]))$ be its Hasse diagram. If there exists an arc from the node V_1 to the node $V_0 = (a, \#_a(wa))$ in $\mathcal{G}' = H(po([wa]))$ then $V_1 \in tail(\mathcal{G})$.*

Proof. Note that $\{V_0\} = occ([wa]) \setminus occ([w])$. By Lemma 1, the node V_1 is a source of V_0 in $\mathcal{G}'$. Suppose that V_1 does not belong to the tail of $\mathcal{G}$. This means that there is a vertex V_2 of the same label as V_1, such that $V_1 \prec_w^+ V_2$. Since V_1 is a source of V_0, we have $V_2 \not\prec_{wa}^+ V_0$. Therefore, $V_2 \not\prec_{wa} V_0$ and $(\ell(V_2), \ell(V_0)) \in ind$. We conclude that there cannot be an arc in $\mathcal{G}'$ between the nodes labeled with $\ell(V_2) = \ell(V_1)$ and $\ell(V_0)$, in particular: between V_1 and V_0. The obtained contradiction proves that V_1 has to be the element of the tail of w, which ends the proof. ■

Algorithm 2: SIMPLE-APPEND($\mathcal{G}$, RT, a)

Input:
- $\mathcal{G}$ – Hasse diagram
- RT – tail($\mathcal{G}$) in reversed order of elements appearances
- a – the letter to be appended to $\mathcal{G}$

```
1  create V– the node of the next occurrence of a (added to G);
2  V.src := {V};
3  V_prev := FIND(RT,a);
4  foreach V' ∈ RT do
5      if (ℓ(V'), ℓ(V)) ∈ dep and V' ∉ V.src then
6          insert an arc V' ⟶ V;
7          add V'.src to V.src;
8  remove V_prev from V.src and RT;
9  foreach V' ∈ RT do
10     remove V_prev from V'.src;
11 insert V to the front of RT;
```

The procedure SIMPLE-APPEND starts with creating a new node V labeled with an action a that will be added to $\mathcal{G}$. We initialize the set of sources of V as an empty set. Then we search the tail for the last occurrence labeled with

a and store it in variable V_{prev}. We use the operation FIND which gets two arguments – a set RT of occurrences of distinct labels and an action a. If it finds an occurrence of a in RT, the node V_{prev} will be removed from proper data structures. If there is no node satisfying the desired condition, the procedure returns $NULL$ value, and nothing has to be removed. This operation can be done in time complexity of $O(\log k)$, where k is the size of alphabet Σ, which limits the size of RT.

After preparing the new node V_{new}, we scan the $tail(\mathcal{G})$. The order of scanning is opposite to the order of adding the elements to the diagram $\mathcal{G}$. We decide, if there should be an arc between the scanned vertex and V. Precisely, we add an arc, if $(\ell(V), \ell(V_{new})) \in dep$ and there is no path from V to V_{new} in the part of $\mathcal{G}$ constructed already. The adopted order, along with the computation of the set of sources during the addition of new arcs, prevent us against the necessity of transitive reduction at the end of the procedure (see Proposition 3.2).

PROPOSITION 3.2 *Let w be a word over a concurrent alphabet (Σ, dep) and let $\mathcal{G} = H(po([w]))$ be its Hasse diagram. Then a node V_1 is a source of a node V_0 and (V_1, V_0) is not an arc in $\mathcal{G}$, if and only if there exists a node V_2 such that V_1 is a source of V_2 and (V_2, V_0) is an arc in $\mathcal{G}$.*

$\Leftarrow$: It follows directly from the definition of a transitive reduction. If we have $V_1 \prec_w^+ V_2 \prec_w^+ V_0$, then the arc (V_1, V_0) is not present in the transitive reduction.

$\Rightarrow$: If V_1 is a source of V_0, then $V_1 \prec_w^+ V_0$, so there is a path in $\mathcal{G}$ that leads from V_1 to V_0.

The last group of instructions is intended to maintain the consistency of the data structure. First, we remove the vertex V_{prev} from the tail structure RT (where it will be replaced by V_{new}) and from the sets of sources of all vertices from $tail(\mathcal{G})$ (as it is no longer in the tail). Finally, we insert the vertex V_{new} into the tail. To preserve the order of elements, the vertex V_{new} is put in front of RT.

The correctness of the Algorithm 2 is straightforward and follows from Proposition 3.1 and Proposition 3.2. Its time complexity is proportional to the square of the size of the alphabet Σ. It is determined by the loop defined in lines 4-7. We iterate through the set of the size limited by the size of the alphabet Σ (denoted by k), check a simple condition and possibly add a new arc, and sum up two sets of sizes at most k.

The sample computation of the Algorithm 2 is presented in Table 1 (see Examples at the end of this paper). In the first column, we can see the linearisations used to build subsequent prefixes of the final diagram, which are presented in the last column. The remaining two columns present the states of data structure during the computation. Note that the superscripted id's and presented in the third column sets del_src are not used by the SIMPLE-APPEND procedure.

3.2. Rebuilding the Hasse diagram

The biggest problem we encounter when using the procedure SIMPLE-APPEND is the unpreparedness of the utilized data structure to removing of nodes from the diagram and its rebuilding. It is easy to see that removing the elements from the border of the diagram (in fact the border of the trace corresponding to the diagram) should not be difficult.

To satisfy this intuition, we have to prepare our data structure for removing terminal nodes and restore the consistent state of the data. We use the whole data structure described at the beginning of this section. It enforces the fundamental changes in the procedure of appending an occurrence to the diagram. The whole pseudocode of extended procedure is presented in the Algorithm 3. It is supported by another algorithm, called REMOVE, which allows for deleting nodes present in the border.

Operations EXTENDED-APPEND and REMOVE

The most significant changes in the append algorithm are visible in the data structure. To fully describe the node, we extend the specification of the node-object and use not only the set *src*, but also its complementary set *del_src*. We also make use of the emphasized predecessor *prev* and the serializer *id*. The reference to the previous node with the same label is used to restore the pointed vertex to the tail. During the addition of the node V that might be deleted from $\mathcal{G}$ in the future, we update the *src* sets of all vertices from RT. To ensure the reversibility of this action, we store in the sets *del_src* the sources that are deleted by the procedure SIMPLE-APPEND. Observe that the field *prev* of an object V in Algorithm 3 plays the role of a local variable v_{prev} in Algorithm 2.

It is worth noting that, according to the definition of a source, the newly created vertex may have sources that are not present in the current tail, but are important from the point of view of the procedure REMOVE (see Table 2 in the Examples at the end of this paper). The extension responsible for this computation is presented in lines 13-20. We scan there all sources of the direct source of the newly added node to find the last occurrence of every action. It is worth mentioning that in the set *del_src* there may exist only one node with the fixed label. At the end of the procedure EXTENDED-APPEND we have to take care of the consistency of the data. Therefore, we delete from *del_src* of V all nodes with the same label as those stored in *src*.

The correctness of the procedure EXTENDED-APPEND follows from the correctness of its base version SIMPLE-APPEND and Proposition 3.2. The most significant, in terms of the computational complexity, is the loop presented in lines 9-20. It performs at most k iterations. In each iteration we may execute another loop with at most k repetitions. This loop performs several operations of removing and inserting with respect to the set of the size of at most $k = |\Sigma|$. Such operations can be done in the time complexity of $\log k$, hence the whole procedure has the time complexity of $O(k^2 \log k)$.

Algorithm 3: EXTENDED-APPEND($\mathcal{G}$, RT, a)

Input:
- $\mathcal{G}$ – Hasse diagram
- RT – tail($\mathcal{G}$) in reversed order of element appearances
- a – the letter to be appended to $\mathcal{G}$

Output: V – the node appended to $\mathcal{G}$

```
1  create V– the node of next occurrence of a in G;
2  if RT = ∅ then
3      V.id := 1;
4  else
5      V.id := FIRST(RT).id+1;
6  V.src := {V};
7  V.del_src := ∅;
8  V.prev :=FIND(RT,a);
9  foreach V' ∈ RT do
10     conditionally insert an arc: V' ⇝ V;
11 remove V.prev from RT and V.src;
12 foreach V' ∈ RT do
13     move V.prev from V'.src to V'.del_src
14 foreach V' ∈ V.del_src do
15     if FIND(V.src, ℓ(V')) ≠ null then
16         remove V' from V.del_src;
17 insert V to front of RT;
18 return V;
```

The last algorithm presented in this section is the procedure REMOVE. It operates, like SIMPLE-APPEND and EXTENDED-APPEND, on the graph $\mathcal{G}$ and its reversed tail RT. The last argument is the node V to be deleted from $\mathcal{G}$. The procedure works properly only if V comes from the border of $\mathcal{G}$. Removing any other node requires recomputing the whole diagram and updating the data structures.

After removing the node V from the diagram $\mathcal{G}$, the previous appearance of the action $\ell(V)$ (if such an action exists in the diagram), stored in V as $prev$, should be inserted to the tail of the modified diagram. All arcs adjacent to V are removed with V. In such a way, we achieve a correct Hasse diagram. We only have to take care of the additional data structures to satisfy all constraints required by the algorithm.

The serializer id, stored in the node $V.prev$, is used to put the node in correct place inside the tail structure RT. Note that all sources of $V.prev$ are stored in src and del_src of this node. We need to move all sources, which are not in RT, into set del_src and these contained in the tail into set src (lines 5-6).

Algorithm 4: Conditional part of the main loop of EXTENDED–APPEND

```
1  if (ℓ(V'), ℓ(V)) ∈ dep and V' ∉ V.src then
2      insert an arc V' ⟶ V;
3      add V'.src to V.src;
4      foreach V'' ∈ V'.del_src do
5          V_d := FIND(V.del_src, ℓ(V''));
6          if V_d ≠ null then
7              if V_d.id < V''.id then
8                  remove V_d from V.del_src;
9                  insert V'' to V.del_src;
10         else
11             insert V'' to V.del_src;
```

The last operation is the fixing of the data stored in the other nodes from RT. We have to move $V.prev$ from del_src to src whenever it is necessary.

The correctness of the procedure REMOVE is straightforward, and follows from the definition of the sets of sources. The time complexity is also obvious. All operations presented in lines 1 and 4-8 may be done in the time complexity of $O(k)$, hence the time complexity of the procedure REMOVE is also $O(k)$.

See Table 2 in Examples at the end of this paper for an example of using the procedures REMOVE and EXTENDED-APPEND to rebuild a Hasse diagram. It is worth observing how the deleted sources are restored after removing the second occurrence of the action a.

4. Generation of all linearisations of the trace

Let τ be a trace over a concurrent alphabet (Σ, dep). Recall that by linearisation of τ we mean a possible ordering of its Hasse diagram nodes $(V_{i_1}, V_{i_2}, \ldots, V_{i_n})$.

For a word w the operation *reverse* (denoted by w^R) is defined in a usual way: $(abcda)^R = adcba$. For a directed graph $\mathcal{G}$ its reverse, denoted by $(\mathcal{G})^R$, is defined as reversing the direction of each arc of $\mathcal{G}$. There is also a straightforward correspondence between nodes of $\mathcal{G}(w)$ and $\mathcal{G}(w^R)$. We can identify the first occurrence of the action a in w with its last occurrence in w^R, the second with last but one, and so on. Formally, the occurrence $\alpha = (a, i)$ in w is identified with $\alpha^R = (a, \#_a(w) - i + 1)$ in w^R. See Fig. 1, shown before, for an example of a directed graph and its reverse.

Remark 1 *Let $\mathcal{G}$ be a directed acyclic graph. Then, directly from the definition: $head(\mathcal{G}) = tail((\mathcal{G})^R)$ and $tail(\mathcal{G}) = head((\mathcal{G})^R)$.*

The following fact will be crucial for the correctness of the algorithms presented further in this section.

Algorithm 5: REMOVE($\mathcal{G}$, RT, V)

Input:
- $\mathcal{G}$ – Hasse diagram
- RT – tail($\mathcal{G}$) in reversed order of elements appearances
- V – the node from BORDER($\mathcal{G}$) to be removed

```
1 remove V from RT;
2 V_p := V.prev;
3 if V_p ≠ null then
4 |   insert V_p to RT with respect to order of id;
5 |   move nodes (V_p.src ∩ RT) from V_p.src to V_p.del_src;
6 |   move nodes (V_p.del_src ∩ RT) from V_p.del_src to V_p.src;
7 |   foreach V' ∈ RT do
8 |   |   move V_p from V'.del_src to V'.src
```

LEMMA 2 *Let w be a word over a concurrent alphabet (Σ, dep) and $\mathcal{G}(w) = H(po([w]))$ be its Hasse diagram. Then the reverse of the Hasse diagram $\mathcal{G}(w)$ is the Hasse diagram of the reverse of w, namely $(\mathcal{G}(w))^R \backsimeq \mathcal{G}(w^R)$. Moreover, V is the minimal vertex in $\mathcal{G}(w)$ if and only if the corresponding vertex V^R is maximal in $\mathcal{G}(w^R)$.*

Proof. Since $pos_w(\alpha) < pos_w(\beta)$ implies that $pos_{w^R}(\beta) < pos_{w^R}(\alpha)$ and vice versa, we have $\alpha \prec_w^+ \beta$ if and only if $\beta^R \prec_{w^R}^+ \alpha^R$. By the definition of the graph $(\mathcal{G}(w))^R$, we have that (V_1, V_2) is and arc in $\mathcal{G}(w)$ if and only if (V_2, V_1) is an arc in $(\mathcal{G}(w))^R$. Therefore, we get the $(\mathcal{G}(w))^R \backsimeq \mathcal{G}(w^R)$ with a vertex $V = (a, i)$ in $(\mathcal{G}(w))^R$ corresponding to the vertex $V^R = (a, \#_a(w) - i + 1)$ in $\mathcal{G}(w^R)$. The second statement is obvious, since $min(occ(w), \prec_w^+) = max(occ(w^R), \prec_{w^R}^+)$. ■

Data structures

The main idea of the algorithms presented in this section is the partition of the current linearisation w of a trace $\tau = [w]$ into the prefix w_{pref} and the suffix w_{suff}. At each step of the computation we process a single action a and move the node corresponding to a from w_{pref} to w_{suff} or vice versa. Therefore, the presented algorithms need some additional data structures.

We use the Hasse diagram $\mathcal{G}$ of the prefix w_{pref} and the Hasse diagram $\mathcal{G}'$ of the reversed suffix $(w_{suff})^R$, called the *dual graph*. The prefix of current linearisation is represented by the list L of nodes of $\mathcal{G}$.

Moreover, we use the notion of a graph border. Recall that for a directed graph $\mathcal{G}$, its border $bord(\mathcal{G})$ is defined as a subset of $tail(\mathcal{G})$ consisting of the nodes having no outgoing arcs. We use the auxiliary procedure COMPUTE-BORDER, which, for a given directed graph $\mathcal{G}$ (more precisely for $tail(\mathcal{G})$), computes $bord(\mathcal{G})$ by choosing nodes with out-degree zero. Since the size of

$tail(\mathcal{G})$ is not greater than k, the procedure COMPUTE-BORDER performs at most k operations.

Procedures MINSUFF and MAXSUFF

The procedure MINSUFF constructs the lexicographically minimal suffix of the current linearisation from the dual graph $\mathcal{G}'$. We process the border of $\mathcal{G}'$ and consecutively choose the node V_{min} with the lexicographically smallest label. The node V_{min} is then removed from $\mathcal{G}'$. A new node with a label $\ell(V_{min})$ is appended to $\mathcal{G}$ and the list L, which is the intermediate state of the created suffix. We use procedures REMOVE and EXTENDED-APPEND described in the previous section. See Algorithm 6 for the technical details.

Algorithm 6: MINSUFF($\mathcal{G}$, RT, $\mathcal{G}'$, RT')

Input:
- $\mathcal{G}$, $\mathcal{G}'$ – Hasse diagrams of current prefix and its reverse
- RT, RT' – reversed $tail(\mathcal{G})$ and $tail(\mathcal{G}')$

Output: L – lexicographically minimal reversed linearisation of $\mathcal{G}'$

```
1 L := ∅;
2 while G' not empty do
3 |   BRD := COMPUTE-BORDER(RT');
4 |   V_min := MIN(BRD);
5 |   REMOVE(G', RT', V_min);
6 |   V_next := EXTENDED-APPEND(G, RT, ℓ(V_min));
7 |   insert V_next to back of L;
8 return L;
```

The correctness of Algorithm 6 follows from Lemma 2. The dual graph $\mathcal{G}'$ is the Hasse diagram of the reversed suffix of current linearisation. We remove only the nodes from the border of $\mathcal{G}'$, hence the structure of Hasse diagram is preserved. Moreover, the method of node selection ensures that the constructed suffix will be lexicographically minimal.

Recall that by k we denote the size of the alphabet Σ and by n – the length of the processed word, in the case considered here the processed suffix. The procedure MINSUFF performs at most n repetitions of the while-loop (lines 2-7). In each iteration, the most time consuming operation is the procedure EXTENDED-APPEND. Therefore, the time complexity of Algorithm 6 is $O(nk^2 \log k)$.

The procedure MAXSUFF can be defined in a similar way as MINSUFF. The only difference appears in the line 4 of the algorithm. We choose the maximal element from the border of the dual graph instead of the minimal one.

Procedures MINLEX and MAXLEX

The procedure MINLEX generates lexicographically minimal linearisation of the trace τ using its Hasse diagram $\mathcal{G}$ and a current linearisation L of τ. First, we process all components of L moving them from the diagram $\mathcal{G}$ to the dual diagram $\mathcal{G}'$. Next, we use the procedure MINSUFF to generate lexicographically minimal suffix of the empty word λ, see Algorithm 7 for details.

The procedure MAXLEX, which computes lexicographically maximal linearisation of τ, is defined in a similar way using the procedure MAXSUFF in line 8 of Algorithm 7. The correctness and the time complexity of MINLEX and MAXLEX (namely $O(nk)$) follow directly from the correctness and the time complexity of MINSUFF and MAXSUFF.

Algorithm 7: MINLEX($\mathcal{G}$, RT, L)

Input:
- $\mathcal{G}$ – Hasse diagram
- RT – reversed $tail(\mathcal{G})$
- L – current linearisation

Output: L – list of nodes of lexicographically minimal linearisation of $\mathcal{G}$

```
1 G' := empty graph;
2 RT' := ∅;
3 while L ≠ ∅ do
4 |   V_l := last element of L;
5 |   remove V_l from L;
6 |   REMOVE(G, RT, V_l);
7 |   EXTENDED-APPEND(G', RT', ℓ(V_l));
8 L := MINSUFF(G, RT, G', RT');
9 return L;
```

Procedures NEXT-LINEARISATION and PREV-LINEARISATION

The procedure NEXT-LINEARISATION allows for browsing the set of all linearisations of the trace τ in the similar way as in the SEPA algorithm (see Knuth, 2005). The main idea of the algorithm is a modification of the suffix of an arbitrary linearisation L_0 so as to obtain L_1 – the next (in lexicographical order) linearisation of τ.

The correctness of the procedure NEXT-LINEARISATION is implied by the following facts, which describe the utilized properties of the border and language-theoretical structure of the lexicographically consecutive linearisations of τ.

Lemma 3 *Let w be a word over concurrent alphabet (Σ,dep) and $\mathcal{G}(w)$ be its Hasse diagram. Then*

$$V_1, V_2 \in bord(\mathcal{G}(w)) \;\Rightarrow\; (\ell(V_1), \ell(V_2)) \in ind.$$

Proof. Let V_1 and V_2 be two nodes belonging to $bord(\mathcal{G}(w))$. Suppose that $(\ell(V_1), \ell(V_2)) \in dep$. Then, by the definition of relation $\prec_w$, we have $V_1 \prec_w V_2$ or $V_2 \prec_w V_1$. Therefore, one of them cannot be maximal in $(occ(w), \prec_w^+)$, hence cannot belong to the border. The obtained contradiction proves that $(\ell(V_1), \ell(V_2)) \in ind$. □

Theorem 1 *Let $u = pau'$ and $v = pbv'$ be two subsequent representatives of a trace τ over a concurrent alphabet (Σ,dep), for $a \neq b \in \Sigma$, and $p, u', v' \in \Sigma^*$. Then $\alpha = (a, \#_a(p) + 1)$ and $\beta = (b, \#_b(p) + 1)$ are two elements of $bord(u'^R a)$ and a, b are subsequent among the set $\ell(bord(u'^R a))$. Moreover, u' is in maxlex normal form and v' is in minlex normal form.*

Proof. Let $u = pau'$ and $v = pbv'$ be two subsequent representatives of the trace τ. Without loss of generality we can assume that $p = \lambda$. Therefore, $\alpha = (a, 1)$ and $\beta = (b, 1)$. The equivalence of u and v implies that $(a, b) \in ind$. Assume that there is another element γ^R of label c in $bord(u^R) = bord(u'^R a)$ such that $a < c < b$. By the definition of the border, γ^R is maximal in u^R. Therefore, the occurrence γ is minimal in u and there exists a linearisation of u starting with c. It is the contradiction with the fact that u and v are lexicographically consecutive linearisations of τ. Hence, actions a and b are indeed subsequent in the set $\ell(bord(u'^R a))$. Suppose that u' is not the maximal element among the representatives of a trace $[u']$. Then there exists a word u'' equivalent with u' and greater than u'. Hence, au'' is equivalent to au' and bv'. Moreover, au'' would be located between au' and bv', which proves the lexicographical maximality of u'. Similarly, we prove that v' is the minimal element among the representatives of a trace $[v']$. □

We start processing the linearisation L_0 by consecutively removing its last element V_l from the tail of the graph $\mathcal{G}$ and inserting a new element with the label $\ell(V_l)$ to the dual graph $\mathcal{G}'$. After each step we examine $bord(\mathcal{G}')$, containing actions, which can be chosen as a next element of the linearisation. We stop this operation if either L_0 is empty or we found in $bord(\mathcal{G}')$ an element V' with a label greater than $\ell(V_l)$.
Due to Theorem 1, the emptiness of L_0 means that there is no possibility of finding a greater (with respect to the lexicographical order) linearisation of the trace τ. In this case the algorithm returns the *null* value. Otherwise, we pick V_n – the smallest element of $bord(\mathcal{G}')$ satisfying the condition $\ell(V_n) > \ell(V_l)$, remove it from $\mathcal{G}'$ and insert to $\mathcal{G}$ the new node with label $\ell(V_n)$. Finally, using the dual graph $\mathcal{G}'$ and the procedure MINSUFF, we generate the lexicographically smallest suffix starting with V_n. See Algorithm 8 for details.

Remark 2 *In algorithms described above the border of the graph does not have to be computed at each step. It can be stored as a local variable and updated*

Algorithm 8: NEXT-LINEARISATION($\mathcal{G}$, RT, L)

Input:
- $\mathcal{G}$ – Hasse diagram
- RT – reversed $tail(\mathcal{G})$
- L – current linearisation

Output: L – list of nodes of the lexicographically next linearisation of $\mathcal{G}$

```
1  G' := empty graph;
2  RT' := ∅;
3  repeat
4  |  if L = ∅ then
5  |  |  return null;
6  |  V := last element of L;
7  |  REMOVE(G, RT, V);
8  |  V' := EXTENDED-APPEND(G', RT', ℓ(V));
9  |  BRD := COMPUTE-BORDER(RT');
10 until MAX(BRD) ≠ V';
11 V_sup := smallest node from BRD greater than V';
12 REMOVE(G', RT', V_sup);
13 V_next := EXTENDED-APPEND(G, RT, ℓ(V_sup));
14 insert V_next to back of L;
15 append MIN-SUFF(G, RT, G', RT') to L;
16 return L;
```

after appending or removing each node. If we append a node V to the graph $\mathcal{G}$, we have to remove from bord($\mathcal{G}$) all direct sources of V. On the other hand, if we remove a node V from $\mathcal{G}$, we must add to bord($\mathcal{G}$) each direct source of V with no outgoing arc. Since every node has at most k sources and tail($\mathcal{G}$) has the size at most k, both approaches have the same time complexity, namely $O(k)$.

Similarly as in the procedure MINSUFF, the repeat-loop in lines (3-10) has the time complexity of $O(nk^2 \log k)$. It performs at most n iterations. In each iteration the most time consuming operation is once more the procedure EXTENDED-APPEND. None of the operations provided in lines 11-15 is more expensive than the procedure MINSUFF. Hence, the time complexity of Algorithm 8 is $O(nk^2 \log k)$.

EXAMPLE 2 *Recall the concurent alphabet from Example 1 and let $\tau = [abacdb]$. Then τ has twelve linearisations:*

$$abacbd, \quad abacdb, \quad abadcb, \quad abcabd, \quad abcadb, \quad abcdab,$$

$$abdacb, \quad abdcab, \quad adbacb, \quad adbcab, \quad dabacb, \quad dabcab.$$

The computation of the linearisation abdacb from the linearisation abcdab using algorithm NEXT-LINEARISATION is shown in Table 3 (see Examples at the end of this paper).

5. Practical implementation

The algorithms described in this paper were implemented in the Java application DiaDem. Its graphical layer utilizes the open source library: *Java Universal Network/Graph Framework* (see O'Madadhain, Fisher and Nelson, 2010). The primary functionality of the application is the graphical presentation of the Hasse diagram of the trace τ over a concurrent alphabet (Σ, dep).

The graph of the concurrent alphabet is displayed in the top right panel and could be defined or redefined by the user. For changing the size of an alphabet one can use two buttons, *Add* and *Del*, visible on the top left panel. The provided table of relation *dep* may be used to set or reset this relation. The dependence between letters can be determined by selecting appropriate fields on this table. After clicking the button *Update*, the resulting concurrent alphabet is visualised in the top right panel and used by the application. The possible size of the alphabet Σ is in the range from 2 to 26, but this limitation arises from the needs of presentation clarity only. Since the size limit of Σ is not enforced by the algorithms it is possible, using the same source code, to remove this restriction by changing the set of used labels (i.e. using letters with indexes a_1, a_2, etc.).

The considered trace τ is defined by an arbitrary representative w, which should be placed in the text field located below the dependence relation graph. The Hasse diagram of $\tau = [w]$ is drawn in the bottom panel. After editing the content of the text field and clicking the button *Update*, the Hasse diagram of τ is either constructed from scratch (if the structure of the alphabet has changed), or the necessary part of the diagram is rebuild (if only the suffix of the processed linearisation has changed). The view of described functionality is depicted in Fig. 2.

Furthermore, application DiaDem allows for previewing the linearisations of the trace τ, which are presented in two ways: as the content of the text field used to define τ, and as the visualisation depicted on the Hasse diagram $\mathcal{G}(\tau)$. The graphical illustration is done by changing the color and thickness of the arcs of $\mathcal{G}(\tau)$. Moreover, if two consecutive elements of the presented linearisation, V_1 and V_2, are not connected by an arc in $\mathcal{G}(\tau)$, a *virtual arc* (V_1, V_2) is added to $\mathcal{G}(\tau)$. Such virtual arc is drawn as a dashed line and is used for the linearisation presentation only, hence it does not affect the Hasse diagram structure. The user turns this additional graphical functionality on or off using the checkbox *Show linearisation*.

Initially, the linearisation given by the user is presented. Using buttons *Next* and *Prev* we can iterate through the set of all linearisations of the trace τ in the lexicographical order. Pressing one of these buttons changes the value displayed in the text field. Moreover, if the visualisation of linearisation is marked to be

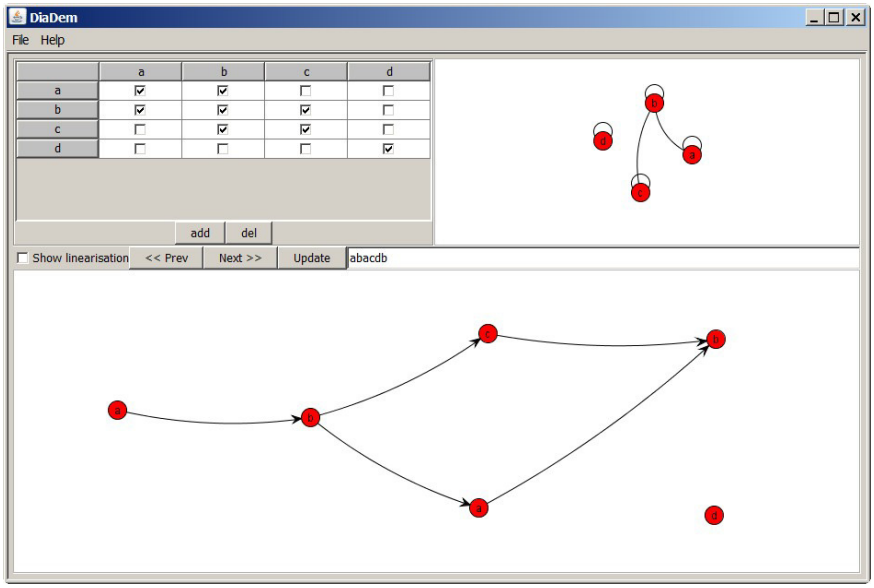


Figure 2. An example screen from the application DiaDem, presenting the basic functionality – a Hasse diagram generated for a given word and dependence relation

shown, the diagram with emphasized (and possibly additional) arcs is redrawn. See Fig. 3 for an example of the linearisation presentation.

The relation structure viewer (top right panel) and the Hasse diagram viewer (bottom pannel) can be used in two different modes: the *transforming* mode, which allows for moving (mouse), rotating (shift+mouse) or transforming (ctrl+ mouse) the whole graph, and the *picking* mode, in which each node could be relocated separately (using drag&drop technique). Such behaviour can be set (for the active panel only) by pressing the `T` key on the keyboard (for transforming mode) or the `P` key (for picking mode). A panel becomes active when the user clicks on it.

The algorithms described in Section 4 process the set of all linearisations of the trace τ by partially destroying and rebuilding its Hasse diagram $\mathcal{G}$. They utilize procedures EXTENDED-APPEND and REMOVE, defined in Section 3. Although such an approach is very useful from the theoretical point of view, DiaDem preserves the structure of the diagram $\mathcal{G}$. The procedures *remove* and *append* are implemented as virtual operations. We have to remember the current prefix L of the processed linearisation and the border *bord* of the dual graph only. The computation of the next linearisation starts with an empty border, which may be updated by the virtual operations *remove* and *append*. Those operations do not affect the structure of the Hasse diagram. It is worth mentioning that

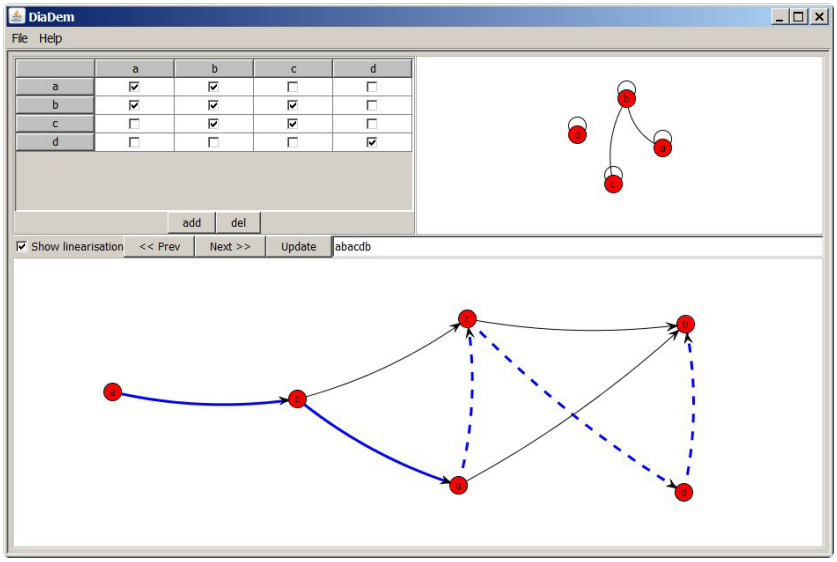


Figure 3. An example screen from the application DiaDem, presenting a linearisation of the trace related to the displayed representative

shifting the border, which demarcates two virtual parts of the graph, is done using the vertical sweep line technique (see Cormen, Leiserson, Rivest and Stein, 2001). The approach, which utilizes the virtual operations on the diagram, does not affect the time complexity of the whole algorithm.

5.1. Other solutions – a short survey

There are several tools, related to the Hasse diagrams, available in the World Wide Web. Most of them are directly related to the theory of partially ordered sets. In most of them one can visualise the structure of computed transitive reduction of a partial order.

We start with a few free tools that are available online. One of the simplest is *JavaScript Lattice Drawing Program* (see Snow, 2006). It shows the visual structure of posets, but is limited to the provided examples only. Any extension would require changes in the program source code.

The *Interactive Poset and Lattice Drawing Java Applet* (see Jipsen, 2006) is another interesting tool for drawing the Hasse diagram of a given poset. The structure of the considered poset must be defined by the web page parameters encoded in html file. The reorganization of the structure of currently drawn poset is impossible.

Another tool is the *Lattice drawing applet* (see Freese, 2004b). It allows for drawing the Hasse diagram of a poset, but is limited to lattices only (i.e.

posets in which any two elements have a unique supremum and a unique infimum). The editing of the structure of a drawn poset is possible, but requires the preparation of file describing the lattice. The webpage provides also some examples of lattices, which are helpful in the preparation of a new example.

The last example of an online tool is The Sage Notebook, see Stein (2004). In contrast to the aforementioned applications, Sage is a very powerful mathematical software system. It allows for defining the poset structure, make some operations on it, for example compute its transitive reduction, and finally draw the graph. The interface is given as a command line, which forces user to learn a very sophisticated syntax. The method of drawing may be fixed by setting proper parameter values. Especially, there are three layouts available. However, there is no possibility of changing the shape of the displayed graph.

The Sage system can be also used as a standalone application. There are also other mathematical environments with similar capabilities. One has to mention Macaulay2, see Grayson et al. (2013), a simple but powerful software system for research in algebraic geometry. A more sophisticated, but also free tool is MuPAD, especially with Combinat package, see Zimmerman (2001). One can not forget about the famous and widely used commercial mathematical softwares, such as Maple (with Posets Package, Stembridge 2009), Mathematica (with Package for Studying Posets, see Greene 2010), and Matlab, see MathWorks (2013) (which seems to have no support for posets operations). It is worth noting that, despite the fact that they use different command sets, all of them are supported by specialised interfaces of the Sage system. However, only Sage has the web interface.

It is worth noting that partial orders are used in the field of formal concept analysis (see Ganter and Wille, 1999) and known as a Hasse Diagram Technique. There is a nice, free tool called DART (see Talete, 2008), which allows for analyzing the previously prepared data, in particular it draws Hasse diagrams for visualization.

As far as we know, there is not only no free useful tool for studying this topic from the point of view of the trace theory, but also none of the existing tools provides the possibility of previewing the linearisations of the visualised poset. The application DiaDem allows for drawing the Hasse diagram of the trace τ and for iterating through the set of all representatives of τ in the lexicographical order. Moreover, since every poset can be generated by a word over a concurrent alphabet (see Mikulski, Piątkowski and Smyczyński, 2011), it could also be used in the case of the arbitrary partial orders. Therefore, the provided visualisation may be utilized in the studies over partially ordered sets.

6. Summary and final remarks

In this paper we presented several algorithms related to the behavior of concurrent systems. In particular, we gave detailed description of two methods for construction of a Hasse diagram (APPEND and EXTENDED-APPEND), together with the FIND procedure. As a complement to the EXTENDED-APPEND,

we provided the algorithm REMOVE. It is worth noting that APPEND may be used to build a Hasse diagram only, while the pair EXTENDED-APPEND and REMOVE allows not only for constructing but also for reconstructing an existing diagram.

The reconstruction process is utilized by the procedures used for browsing all linearisations of a presented poset. At the end of Section 4 we introduced algorithms NEXT-LINEARISATION and PREV-LINEARISATION which, for a given linearisation, allow for computing the next one in the lexicographical order (respectively the previous one). Those two methods make use of the procedures MINSUFF and MAXSUFF, which append the lexicographically minimal or maximal possible suffix to a given prefix of a linearisation. By applying them to the empty prefix we obtain the procedures MINLEX and MAXLEX that construct minimal and maximal linearisations.

We investigate these algorithms from the point of view of the causal dependencies of the single atomic actions. Namely, we provide the efficient tools for online construction and modification of the Hasse diagrams, which describe the causal partial order of actions. The described algorithms may be, in a straightforward way, adapted for more complex concurrency models, see for example Mikulski and Koutny (2011).

The natural further direction of investigations in this area is to provide an algorithm, which generates all nonequivalent traces of a given length. An important property of such generation would be the uniqueness of the trace causal relation structure. Such an algorithm should output the diagram of a fixed shape only once during the whole generation procedure. Moreover, we plan to extend functionality of the application by adding modules related to chains and antichains.

There are also some possible improvements of the provided application. From the theoretical point of view, one can try to improve the time complexity of the operation EXTENDED-APPEND. The cost of such improvement may be a worse time complexity of the operation REMOVE (acceptable up to $O(k^2)$). Moreover, a better method of the graph nodes positioning may be implemented.

Acknowledgements

We would like to thank the anonymous reviewers for their constructive comments.

The research described in this paper has been partially supported by the National Science Center under the grant No. 2011/01/B/ST6/01477.

References

Aho, A. V., Garey, M. R. and Ullman, J. D. (1972) The transitive reduction of a directed graph. *SICOMP: SIAM Journal on Computing*, 1, 131–137.

Cartier, P. and Foata, D. (1969) Problèmes combinatoires de commutation et réarrangements, **LNM 85**. Springer, Berlin.
Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. (2001) *Introduction to Algorithms*. MIT Press, 2 ed.
Diekert, V. and Rozenberg, G., eds. (1995) *The Book of Traces*. World Scientific, Singapore.
Dilworth, R. P. (1950) A decomposition theorem for partially ordered sets. *Annals of Mathematics*, **51**(1), 161–166.
Eve, J. and Kurki-Suonio, R. (1977) On computing the transitive closure of a relation. *Acta Informatica*, **8**, 303–314.
Freese, R. (2004a) Automated lattice drawing. In: *International Conference on Formal Concept Analysis (ICFCA)*, LNCS 2, 112–127.
Freese, R. (2004b) Lattice drawing applet. http:/latdraw.org. Retrieved on 12.09.13.
Ganter, B. and Wille, R. (1999) *Formal Concept Analysis: Mathematical Foundations*. Springer.
Garg, A. and Tamassia, R. (1994) Advances in graph drawing. In: *CIAC: Italian Conference on Algorithms and Complexity*. Springer, 12–21.
Gastin, P. (1990) Infinite traces. In: *Semantics of Systems of Concurrent Processes*. LNCS 469, Springer 277–308.
Graham, R. L., Grötschel, M. and Lovász, L., eds. (1995) *Handbook of Combinatorics (vol. 2)*. MIT Press, Cambridge, MA, USA.
Grayson, D., Stillman, M. and Eisenbud, D. (2013) Macaulay2 software system v. 1.6. http://www.math.uiuc.edu/Macaulay2/. Retrieved on 12.09.13.
Greene, C. (2010) Mathematica package for studying posets v. 3.0. http://www.haverford.edu/math/cgreene/posets.html. Retrieved on 12.09.13.
Hopcroft, J. E. and Ullman, J. D. (1979) *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.
IBM (2008) Infosphere platform. http://www-01.ibm.com/software/data/infosphere/. Retrieved on 12.09.13.
Jipsen, P. (1999) Interactive Poset and Lattice Drawing Java Applet. http://www1.chapman.edu/~jipsen/gap/posets.html. Retrieved on 12.09.13.
Knuth, D. E. (2005) *The Art of Computer Programming: Volume 4, Fascicle 3. Generating All Combinations and Partitions*. Addison-Wesley.
MathWorks (2013) MATLAB – the language for technical computing v. 5.01. http://www.mathworks.com/products/matlab/. Retrieved on 12.09.13.
Mazurkiewicz, A. (1977) Concurrent program schemes and their interpretations. Daimi report pb-78, Aarhus University.
Mikulski, Ł. (2008) Projection representation of Mazurkiewicz traces. *Fundamenta Informaticae*, 85, 399–408.
Mikulski, Ł. and Koutny, M. (2011) Hasse diagrams of combined traces. Technical report cs-tr-1301, Newcastle University.
Mikulski, Ł. and Piątkowski, M. (2012) Diadem – Hasse diagram demon-

strator. http://folco.mat.umk.pl/DiaDem/. Retrieved on 12.09.13.

Mikulski, Ł., Piątkowski, M. and Smyczyński, S. (2011) Algorithmics of posets generated by words over partially commutative alphabets. In: J. Holub, and J. Žďárek, eds. *Proceedings of the Prague Stringology Conference 2011*. Czech Technical University in Prague, Prague, 209–219.

O'Madadhain, J., Fisher, D. and Nelson, T. (2010) JUNG: Java Universal Network/Graph Framework v. 2.0.1.
http://jung.sourceforge.net/. Retrieved on 12.09.13.

Petri, C. A. (1962) *Kommunikation mit Automaten*. Ph.D. thesis, University of Bonn, Bonn, Germany.

Poutré, J. A. L. and van Leeuwen, J. (1987) Maintenance of transitive closures and transitive reductions of graphs. In: H. Göttler and H. J. Schneider, eds., *Proc. of Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS 314, Springer, 106–120.

Purdom, P. W. (1970) A transitive closure algorithm. *BIT*, 10, 76–94.

Rozenberg, G. and Salomaa, A.. eds., (1997) *Handbook of Formal Languages, vol. 1: Word, Language, Grammar*. Springer-Verlag New York, Inc.

Snow, J. (2006) JavaScript Lattice Drawing Program.
http://estrada.cune.edu/facweb/john.snow/resources/drawLat.html.
Retrieved on 12.09.13.

Stein, W. (2004) Sage mathematical software system.
http://www.sagemath.org/doc/reference/index.html. Retrieved on 12.09.13.

Stembridge, J. (2009) Maple posets package v. 2.4.
http://www.math.lsa.umich.edu/~jrs/maple.html. Retrieved on 12.09.13.

Stevenson, W. (2007) *Operations Management*. McGraw-Hill/Irwin, 9th ed.

Szpilrajn, E. (1930) Sur l'extension de l'ordre partiel. *Fundamenta Mathematicae*, **16** (1), 386–389.

Talete (2008) Dart (decision analysis by ranking techniques).
http://www.talete.mi.it/products/dart_description.htm. Retrieved on 12.09.13.

Zimmerman, P. (2001) MuPAD-combinat – algebraic combinatorics package for MuPAD. http://mupad-combinat.sourceforge.net/. Retrieved on 12.09.13.

Examples

In this section we provide a few examples to demonstrate the running details of the presented algorithms. We start with an illustration of Hasse diagram generation for a given concurrent word using algorithm EXTENDED-APPEND (see Table 1). Next, we show how such a diagram can be transformed using procedures REMOVE and EXTENDED-APPEND (see Table 2). Finally we show how the structure of Hasse diagram is utilized to compute next (in lexicographical order) linearisation of related concurrent word (see Table 2).

Current prefix and reversed tail	Sources	Deleted sources	Hasse diagram
$\boldsymbol{a}$ $\left[a_1^{(1)}\right]$	$a_1^{(1)} : [a_1^{(1)}]$	$a_1^{(1)} : \emptyset$	$a_1^{(1)}$
a$\boldsymbol{b}$ $\left[b_1^{(2)}, a_1^{(1)}\right]$	$b_1^{(2)} : [a_1^{(1)}, b_1^{(2)}]$ $a_1^{(1)} : [a_1^{(1)}]$	$b_1^{(2)} : \emptyset$ $a_1^{(1)} : \emptyset$	$a_1^{(1)} \longrightarrow b_1^{(2)}$
ab$\boldsymbol{a}$ $\left[a_2^{(3)}, b_1^{(2)}\right]$	$a_2^{(3)} : [b_1^{(2)}, a_2^{(3)}]$ $b_1^{(2)} : [b_1^{(2)}]$	$a_2^{(3)} : \emptyset$ $b_1^{(2)} : [a_1^{(1)}]$	$a_1^{(1)} \longrightarrow b_1^{(2)} \nearrow a_2^{(3)}$
aba$\boldsymbol{c}$ $\left[c_1^{(4)}, a_2^{(3)}, b_1^{(2)}\right]$	$c_1^{(4)} : [b_1^{(2)}, c_1^{(4)}]$ $a_2^{(3)} : [b_1^{(2)}, a_2^{(3)}]$ $b_1^{(2)} : [b_1^{(2)}]$	$c_1^{(4)} : [a_1^{(1)}]$ $a_2^{(3)} : \emptyset$ $b_1^{(2)} : [a_1^{(1)}]$	$a_1^{(1)} \longrightarrow b_1^{(2)} \nearrow a_2^{(3)}$ $\qquad\qquad\quad \searrow c_1^{(4)}$
abac$\boldsymbol{d}$ $\left[d_1^{(5)}, c_1^{(4)}, a_2^{(3)}, b_1^{(2)}\right]$	$d_1^{(5)} : [d_1^{(5)}]$ $c_1^{(4)} : [b_1^{(2)}, c_1^{(4)}]$ $a_2^{(3)} : [b_1^{(2)}, a_2^{(3)}]$ $b_1^{(2)} : [b_1^{(2)}]$	$d_1^{(5)} : \emptyset$ $c_1^{(4)} : [a_1^{(1)}]$ $a_2^{(3)} : \emptyset$ $b_1^{(2)} : [a_1^{(1)}]$	$a_1^{(1)} \longrightarrow b_1^{(2)} \nearrow a_2^{(3)}$ $\qquad\qquad\quad \searrow c_1^{(4)}$ $d_1^{(5)}$
abacd$\boldsymbol{a}$ $\left[a_3^{(6)}, d_1^{(5)}, c_1^{(4)}, b_1^{(2)}\right]$	$a_3^{(6)} : [b_1^{(2)}, a_3^{(6)}]$ $d_1^{(5)} : [d_1^{(5)}]$ $c_1^{(4)} : [b_1^{(2)}, c_1^{(4)}]$ $b_1^{(2)} : [b_1^{(2)}]$	$a_3^{(6)} : \emptyset$ $d_1^{(5)} : \emptyset$ $c_1^{(4)} : [a_1^{(1)}]$ $b_1^{(2)} : [a_1^{(1)}]$	$a_2^{(3)} \longrightarrow a_3^{(6)}$ $a_1^{(1)} \longrightarrow b_1^{(2)} \nearrow$ $\qquad\qquad\quad \searrow c_1^{(4)}$ $d_1^{(5)}$

Table 1. The example of the Hasse diagram generation for the word *abacda*. The structure *Deleted sources* is used only by the algorithm EXTENDED-APPEND.

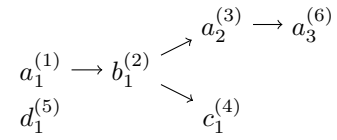
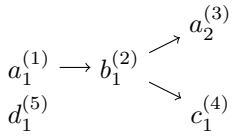
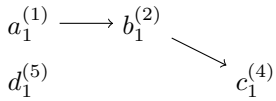
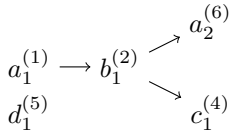
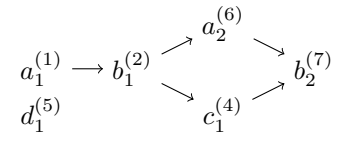
Performed operation and reversed tail	Sources	Deleted sources	Hasse diagram
$\mathcal{G}(abacda)$ $\left[a_3^{(6)}, d_1^{(5)}, c_1^{(4)}, b_1^{(2)}\right]$	$a_3^{(6)} : [b_1^{(2)}, a_3^{(6)}]$ $d_1^{(5)} : [d_1^{(5)}]$ $c_1^{(4)} : [b_1^{(2)}, c_1^{(4)}]$ $b_1^{(2)} : [b_1^{(2)}]$	$a_3^{(6)} : \emptyset$ $d_1^{(5)} : \emptyset$ $c_1^{(4)} : [a_1^{(1)}]$ $b_1^{(2)} : [a_1^{(1)}]$	
REMOVE$\left(a_3^{(6)}\right)$ $\left[d_1^{(5)}, c_1^{(4)}, a_2^{(3)}, b_1^{(2)}\right]$	$a_2^{(3)} : [b_1^{(2)}, a_2^{(3)}]$ $d_1^{(5)} : [d_1^{(5)}]$ $c_1^{(4)} : [b_1^{(2)}, c_1^{(4)}]$ $b_1^{(2)} : [b_1^{(2)}]$	$a_2^{(3)} : \emptyset$ $d_1^{(5)} : \emptyset$ $c_1^{(4)} : [a_1^{(1)}]$ $b_1^{(2)} : [a_1^{(1)}]$	
REMOVE$\left(a_2^{(3)}\right)$ $\left[d_1^{(5)}, c_1^{(4)}, b_1^{(2)}, a_1^{(1)}\right]$	$a_1^{(1)} : [a_1^{(1)}]$ $d_1^{(5)} : [d_1^{(5)}]$ $c_1^{(4)} : [b_1^{(2)}, c_1^{(4)}, a_1^{(1)}]$ $b_1^{(2)} : [b_1^{(2)}, a_1^{(1)}]$	$a_1^{(1)} : \emptyset$ $d_1^{(5)} : \emptyset$ $c_1^{(4)} : \emptyset$ $b_1^{(2)} : \emptyset$	
APPEND$\left(a\right)$ $\left[a_2^{(6)}, d_1^{(5)}, c_1^{(4)}, b_1^{(2)}\right]$	$a_2^{(6)} : [b_1^{(2)}, a_2^{(6)}]$ $d_1^{(5)} : [d_1^{(5)}]$ $c_1^{(4)} : [b_1^{(2)}, c_1^{(4)}]$ $b_1^{(2)} : [b_1^{(2)}]$	$a_2^{(6)} : \emptyset$ $d_1^{(5)} : \emptyset$ $c_1^{(4)} : [a_1^{(1)}]$ $b_1^{(2)} : [a_1^{(1)}]$	
APPEND$\left(b\right)$ $\left[b_2^{(7)}, a_2^{(6)}, d_1^{(5)}, c_1^{(4)}\right]$	$a_2^{(6)} : [b_1^{(2)}, a_2^{(6)}]$ $d_1^{(5)} : [d_1^{(5)}]$ $c_1^{(4)} : [c_1^{(4)}]$ $b_2^{(7)} : [b_2^{(7)}, a_2^{(6)}, c_1^{(4)}]$	$a_2^{(6)} : [b_1^{(2)}]$ $d_1^{(5)} : \emptyset$ $c_1^{(4)} : [a_1^{(1)}, b_1^{(2)}]$ $b_2^{(7)} : \emptyset$	

Table 2. The example of the Hasse diagram modification using operations EXTENDED-APPEND and REMOVE.

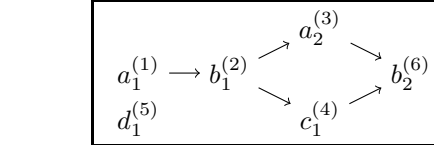
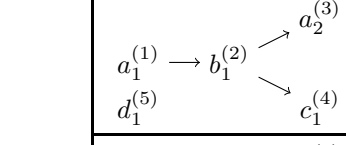
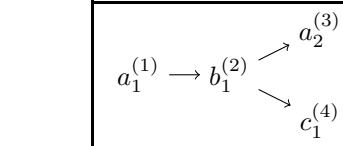
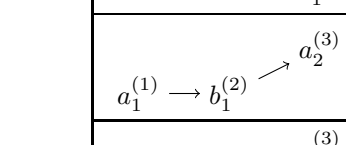
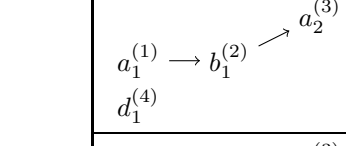
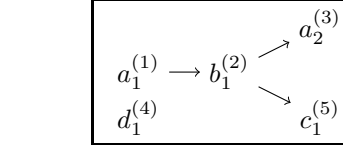
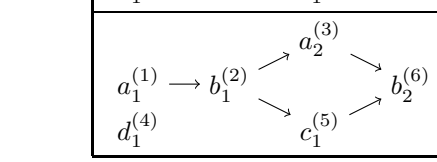
Hasse diagram	Prefix and reversed tail	Suffix and reversed tail	Dual graph
	$abacdb$ $\left[\mathbf{b_2^{(6)}}, \mathbf{d_1^{(5)}}, c_1^{(4)}, a_2^{(3)}\right]$	λ $[\emptyset]$	
	$a\,b\,a\,c\,d$ $\left[\mathbf{d_1^{(5)}}, \mathbf{c_1^{(4)}}, \mathbf{a_2^{(3)}}, b_1^{(2)}\right]$	b $\left[\mathbf{b_1^{(1)}}\right]$	$b_1^{(1)}$
	$a\,b\,a\,c$ $\left[\mathbf{c_1^{(4)}}, \mathbf{a_2^{(3)}}, b_1^{(2)}\right]$	$d\,b$ $\left[\mathbf{d_1^{(2)}}, \mathbf{b_1^{(1)}}\right]$	$b_1^{(1)}$ $d_1^{(2)}$
	$a\,b\,a$ $\left[\mathbf{a_2^{(3)}}, b_1^{(2)}\right]$	$c\,d\,b$ $\left[\mathbf{c_1^{(3)}}, \mathbf{d_1^{(2)}}, b_1^{(1)}\right]$	$b_1^{(1)} \longrightarrow c_1^{(3)}$ $d_1^{(2)}$
	$a\,b\,a\,d$ $\left[\mathbf{d_1^{(4)}}, \mathbf{a_2^{(3)}}, b_1^{(2)}\right]$	$c\,b$ $\left[\mathbf{c_1^{(3)}}, b_1^{(1)}\right]$	$b_1^{(1)} \longrightarrow c_1^{(3)}$
	$a\,b\,a\,d\,c$ $\left[\mathbf{c_1^{(5)}}, \mathbf{d_1^{(4)}}, \mathbf{a_2^{(3)}}, b_1^{(2)}\right]$	b $\left[\mathbf{b_1^{(1)}}\right]$	$b_1^{(1)}$
	$a\,b\,a\,d\,c\,b$ $\left[\mathbf{b_2^{(6)}}, c_1^{(5)}, \mathbf{d_1^{(4)}}, a_2^{(3)}\right]$	λ $[\emptyset]$	

Table 3. The exemplary computation of the algorithm NEXT-LINEARISATION for the trace τ from Example 2.