Decidability of split equivalence

Y. Abramson and A. Rabinovich
Department of Computer Science
The Sackler School of Exact Sciences
Tel Aviv University
e-mails: yotam@math.tau.ac.il, rabino@math.tau.ac.il

Abstract

We show that the following problem is decidable: given expressions E_1 and E_2 constructed from variables by the regular operations and shuffle, is the identity $E_1 = E_2$ true for all instantiations of its variables by strings?
Our proof uses the notations developed in the causal approach to concurrency. As a byproduct we obtain decidability of similar equivalence for other formalisms. In particular, we prove decidability of split equivalence for Petri nets.
Our paper also provides an alternative proof for a characterization of split equivalence recently given by W.Vogler.

1 Introduction

1.1 Equivalence between open expressions

We deal with some problem inspired by the following classical problem:

Problem 1 Equivalence of open regular expressions under instantiation of variables by languages.
Input: two expressions E_1 and E_2, constructed from variables, constants and regular operations - concatenation (;), union (+) and iteration (*).
Question: is the identity $E_1 = E_2$ valid for every instantiation of the variables by languages?

For example, the identity $(X + Y)^* = (X^*Y^*)^*$ is true because for all languages L_1 and L_2, the languages $(L_1 + L_2)^*$ and $(L_1^* L_2^*)^*$ are the same. In this case we say that E_1 and E_2 are *language equivalent* (notation $E_1 \sim_{lang} E_2$).
Problem 1 is decidable; a "folk" theorem (see [11], Chap. 3, Ex. 14) says that such an identity is valid iff it is valid when variables are instantiated by single letters. Applying this theorem to the example given above, we can see that the identity $(X+Y)^* = (X^*Y^*)^*$

is valid because the regular variable-free identity $(x+y)^* = (x^*y^*)^*$ is valid. Checking a variable-free regular equation is a matter of comparing between finite automata.
A related problem is:

Problem 2 Equivalence of open regular expressions under instantiation of variables by strings.
Input: two expressions E_1 and E_2, constructed from variables, constants and regular operations - concatenation (;), union (+) and iteration (∗).
Question: is the identity $E_1 = E_2$ valid for every instantiation of the variables by strings?

Consider again the identity $(X+Y)^* = (X^*Y^*)^*$. This equation is valid for all instantiation of variables by strings: for all strings s_1 and s_2, the languages $(s_1+s_2)^*$ and $(s_1^*s_2^*)^*$ are the same. We say that the two expressions are *string equivalent* (notation $E_1 \sim_{str} E_2$).

The fact that we obtain the same answer for the validity of the above identity under language and string instantiation is not surprising. If an equation is valid under language instantiation, then in particular it is valid under instantiation of variables by single-word languages, that are strings. But is the converse implication also true?

A brief look at the "folk" theorem gives us the positive answer: if an equation is valid under string interpretation, then it is valid under single-letter interpretation, and thus it is valid for languages as well. So as far as only regular operations are involved, the **language interpretation validity problem** is equivalent to the **string interpretation validity problem**.

The "folk" theorem fails if we allow the expressions to contain other operations, such as shuffle (notation $\|$) or intersection (notation $\cap$). For example: an equation $X \parallel Y = XY + YX$ is valid under instantiation of X and Y by single letters x and y, because $x \parallel y = xy + yx$. However, when X is instantiated by x_1x_2 and Y instantiated by y_1y_2, we get the equation $x_1x_2 \parallel y_1y_2 = x_1x_2y_1y_2 + y_1y_2x_1x_2$, which is incorrect, since the word $x_1y_1x_2y_2$ belongs only to the left side.

Moreover, when we allow the shuffle operation, an equation might be valid under all instantiations of variables by strings, and still not valid under instantiation of variables by languages. For example (given in [13]), an equation $XY \parallel YZ = XY \parallel YZ + XYZ \parallel Y$ is valid when X, Y and Z range over strings; it is not valid when they range over languages. Take $X = \{x\}, Y = \{ab, cd\}, Z = \{z\}$.

Let us consider problems 1 and 2 for shuffle regular expressions:

Problem 3 Equivalence of open shuffle regular expressions under instantiation of variables by languages.
Input: two expressions E_1 *and* E_2*, constructed from variables, constants, regular operations - concatenation (;), union (+) and iteration (*) - and the shuffle operation (*$\|$*).*
Question: is the identity $E_1 = E_2$ *valid for every instantiation of the variables by languages?*

Problem 4 Equivalence of open shuffle regular expressions under instantiation of variables by strings.
Input: two expressions E_1 *and* E_2*, constructed from variables, constants, regular operations - concatenation (;), union (+) and iteration (*) - and the shuffle operation (*$\|$*).*
Question: is the identity $E_1 = E_2$ *valid for every instantiation of the variables by strings?*

In addition to these problems, we mention two more trivial problems:

Problem 5 Equivalence of open shuffle regular expressions under instantiation of variables by n-languages.
Input: two expressions E_1 *and* E_2*, constructed from variables, constants, regular operations - concatenation (;), union (+) and iteration (*) - and the shuffle operation (*$\|$*).*
Question: is the identity $E_1 = E_2$ *valid for every instantiation of the variables by languages which contain strings of length* $\leq n$*?*

For two expressions E_1 and E_2 which are equivalent under instantiation of variables by n-languages, we use the notation $E_1 \sim^n_{lang} E_2$

Problem 6 Equivalence of open shuffle regular expressions under instantiation of variables by n-strings.
Input: two expressions E_1 *and* E_2*, constructed from variables, constants, regular operations - concatenation (;), union (+) and iteration (*) - and the shuffle operation (*$\|$*).*
Question: is the identity $E_1 = E_2$ *valid for every instantiation of the variables by strings of length* $\leq n$*?*

For two expressions E_1 and E_2 which are equivalent under instantiation of variables by n-strings, we use the notation $E_1 \sim^n_{str} E_2$
We also mention the following theorem of Gischer [4]:

Theorem 1.1 *Let* E_1 *and* E_2 *be shuffle-regular expressions that contains the variables* $V_1, V_2 \ldots V_n$*. Then* E_1 *and* E_2 *are language-equivalent iff they are equivalent under the following interpretation of the variables:*
$V_i \leftrightarrow \{v^S_{i,j} v^F_{i,j} : j \in Nat\}$

In the above theorem, we used the form $V_i \leftrightarrow L$ to express a language interpretation that assigns the language L to the variable V_i. Note that the language assigned to variable V_i is a language over the infinite alphabet $\{v^F_{i,j}, v^S_{i,j} : j \in Nat\}$. It contains all the strings $v^S_{i,j}, v^F_{i,j}$.

This theorem states that $\sim^2_{lang}$ and $\sim_{lang}$ coincide. Therefore $\sim_{lang}=\sim^n_{lang}$ for all $n > 1$, and we have no hierarchy. However, such hierarchy *does* exist for $\sim_{str}$. Van-Glabbeek and Vaandrager [12] have shown that the equivalence $\sim_{str}$ is strictly finer than $\sim^2_{str}$. Moreover, $\sim^{n+1}_{str}$ is strictly finer than $\sim^n_{str}$ for any n. Hence:

$$\sim^1_{str} \supsetneq \sim^2_{str} \supsetneq \sim^3_{str} \supsetneq \dots \supsetneq \sim_{str} \supsetneq \sim_{lang} = \sim^2_{lang}$$

For example, the expressions $XY \parallel YZ + X \parallel (Y;(Y \parallel Z))$ and $XYZ \parallel Y + X \parallel (Y;(Y \parallel Z))$ are string-2 but not string-3 equivalent. To verify that they are string-2 equivalent we must check all options; to see that they are not string-3 equivalent, instantiate $X = x_1x_2x_3, Y = y_1y_2y_3, Z = z_1z_2z_3$. The word $x_1y_1x_2x_3y_2y_1y_3z_1y_2y_3z_2z_3$ does not belong to the right side.
At the end of this paper we give our examples for distinguishing between $\sim^n_{str}$ and $\sim^{n+1}_{str}$.

1.2 Decidability

Problem 3 was proved decidable by Meyer and Rabinovich [6]. Their proof uses Theorem 1.1. The instantiation suggested in this theorem is, of course, infinite and does not give the decidability result. In [6] it is shown that it's enough to instantiate V_i by the finite language $\Sigma^{k+1}_{j=1} V^S_{i,j} V^F_{i,j}$, where k is the **shuffle width** of the expressions. For example, the shuffle width of $(X \parallel Y) \parallel Z$ is 2, so in order to check the identity $(X \parallel Y) \parallel Z = X \parallel (Y \parallel Z)$ it is enough to check for the instantiation $X = \{x^S_1x^F_1, x^S_2x^F_2, x^S_3x^F_3\}, Y = \{y^S_1y^F_1, y^S_2y^F_2, y^S_3y^F_3\}, Z = \{z^S_1z^F_1, z^S_2z^F_2, z^S_3z^F_3\}$.

Problem 4 was investigated by Vogler [13], which states that its decidability is still an open problem. The main result of this paper is that problem 4 is decidable. We give the following algorithm for checking a string-equivalence between two expressions:

Algorithm for deciding string equivalence : Given two expressions E_1 and E_2, in order to decide if they are string-equivalence, check string-n+1 equivalence, (i.e. instantiate any variable X_i by the string $x^1_i, x^2_i \dots x^{n+2}_i$), where n is the maximum shuffle width of the expressions. Then check equivalence between the two resulting variable-free expressions.

Say, for example, that we would like to verify that $XY \parallel YZ$ is string-equivalent to $XY \parallel YZ + XYZ \parallel Y$. The shuffle width of both expressions is 2, we should only check

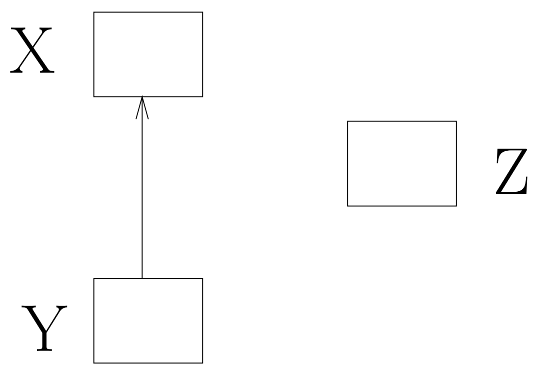


Figure 1: The partial order semantics of the expression $XY \parallel Z$.

split-3 equivalence. We do that by the instantiation $X = x_1x_2x_3, Y = y_1y_2y_3, Z = z_1z_2z_3$. We get the following variable-free equation:
$x_1x_2x_3y_1y_2y_3 \parallel y_1y_2y_3z_1z_2z_3 = x_1x_2x_3y_1y_2y_3 \parallel y_1y_2y_3z_1z_2z_3 + x_1x_2x_3y_1y_2y_3z_1z_2z_3 \parallel y_1y_2y_3$
Checking this equation is a matter of comparing between finite automata.

1.3 Main technique of the paper

Approaches to the semantics of concurrent systems may be divided into two main groups: interleaving and partial order. In the interleaving approach, only the temporal behavior of the events of a run is observable; in the partial order approach, 'causal dependencies' between events are considered.
Consider for example the expression $xy \parallel z$. The interleaving approach assigns to this expression a set of strings: $\{zxy, xzy, xyz\}$; the partial order semantics assigns to this expression the labeled partial order which is shown in figure 1. In this figure, boxes represent elements. If an element v precedes element v', then there is an arrow from v to v'. Labels are put near the corresponding boxes.

A **labled Partial Order (l.p.o.)** is a partial order with a labeling function. A **partially ordered multiset** (or a **pomset**) (see [9]) is an isomorphism class of lpo's. Pomset semantics of expressions gives a set of pomsets (pomset-language) to every expression. Several examples for assigning expressions with pomset-languages are given in figure 2.

In this paper we take a general approach and make our proofs on pomsets, instead of proving particularly on expressions. Apart from being more convenient, the proof on the abstract level of pomsets gives us, as a byproduct, results for other formalisms. In particular, the decidability of split-equivalence for Petri nets is obtained.

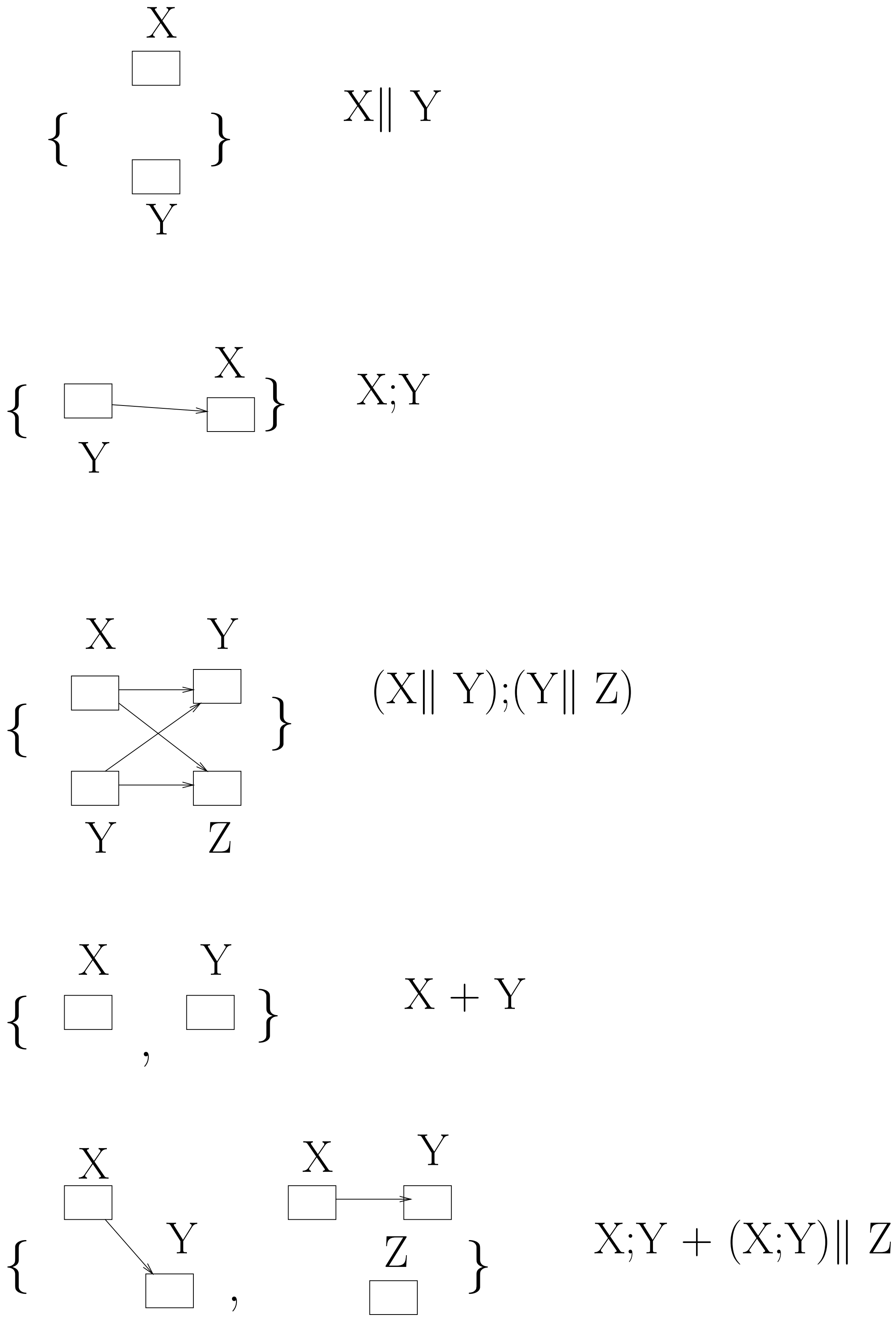


Figure 2: Expressions with their associated pomset-languages

The rest of this paper is organized as follows. In section 2 we give some preliminaries from the theory of pomsets and describe the connections between pomsets and systems such as Petri nets and shuffle-regular expressions. Section 3 states the main results.
Section 4 introduces the notion of a **distinctive word**, which allows to distinguish between pomsets. Section 5 shows the existence of a distinctive word by giving a way to construct such a word from a given pomset. The proof of the main theorem is given in sections 6, 7, 8 and 9. The swap equivalence of pomsets, introduced by Vogler [13], plays an important role in our proofs.
Section 10 shows the optimality of our main theorem, and gives another proof of the theorem due to van Glabbeek and Vaandrager ([12]) that $split_{n+1}$ equivalence is strictly finer than $split_n$ equivalence.

Section 11 concludes the paper with some remarks and discussion of related topics.

2 Preliminaries

2.1 Pomsets - basic definitions

Definition 2.1 *(pomset) [9] A* **partially ordered set (poset)** $P = (V_P, <_P)$ *consists of a finite set* V_P *(of events) and an irreflexive, transitive relation* $<_P$ *on* V_P*. Two events* e *and* e' *are* **concurrent** *if neither* $e <_P e'$ *nor* $e' <_P e$*. A* **labled partial order (l.p.o)** *over a set* Σ *of labels is a partial order and a labeling function* $lab_P : V_P \to \Sigma$*. Two lpo's are isomorphic if there is a label-preserving isomorphism between their partial orders. A* **Partially ordered multiset (pomset)** *is an isomorphism class of lpo's. When clear from the context, we will abuse terminology and will not distinguish between a labled partial order and its corresponding pomset.*

Definition 2.2 *The* width *of a pomset* P *is the maximum number of multually concurrent events in* V_P*.*

A *string* $A_1 A_2 \dots A_n$ will be identified with the pomset $P = (V_P, <_P, lab_P)$, where:
$V_P = v_1, v_2 \dots v_n$,
$v_i <_P v_j$ iff $i < j$ and
$lab_P(v_i) = A_i$.

Definition 2.3 *A* **pomset language** *is a set of pomsets.*

Definition 2.4 *(Interval poset) A poset* P *is called an* **interval poset** *if there is a linearly ordered set* $\langle A, <_A \rangle$*, and two function* $G_s : V_P \to A$ *and* $G_f : V_P \to A$*, such that:*

1. $\forall v \in V_P . G_s(v) <_A G_f(v)$
2. $\forall v_1 v_2 . v_1 <_P v_2$ *iff* $G_f(v_1) <_A G_s(v_2)$.

An **interval pomset** *is a pomset whose poset is an interval poset.*

Instead of the linear order A we can use natural numbers in Definition 2.4, and the notion of interval poset will stay the same.
The functions G_s and G_f can be interpreted as assigning to each event v an *interval* $[G_s(v), G_f(v)]$.
Note that an interval poset P can be represented by a string over $V_P \times \{S, F\}$. Consider, for example, the interval pomset in figure 3. The figure shows the pomset and a set of corresponding intervals. The intervals can be represented by the following string:
$\langle v_1, S\rangle\langle v_4, S\rangle\langle v_1, F\rangle\langle v_6, S\rangle\langle v_2, S\rangle\langle v_4, F\rangle\langle v_5, S\rangle\langle v_2, F\rangle\langle v_3, S\rangle\langle v_5, F\rangle\langle v_6, F\rangle\langle v_3, F\rangle$. In this representation, a letter $\langle v, S\rangle$ (respectively $\langle v, F\rangle$) denotes the start (respectively finish) of an event.

Therefore we have the following definition.

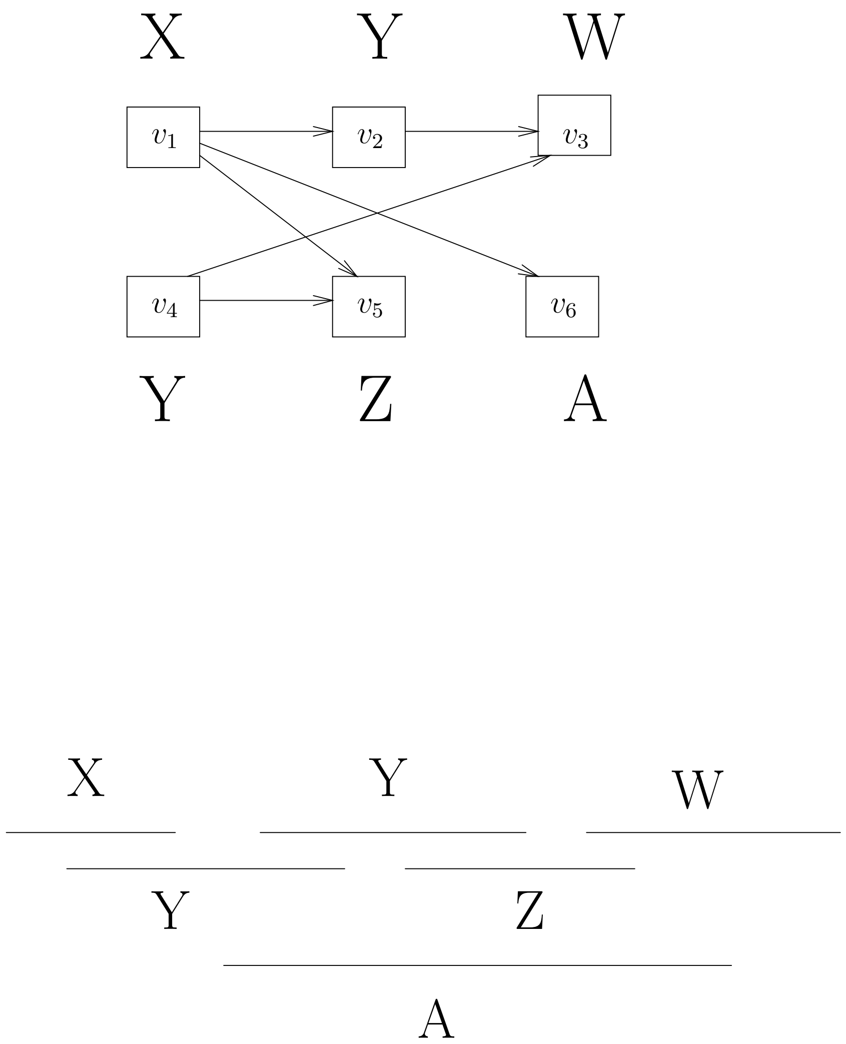


Figure 3: An interval pomset with its intervals.

Definition 2.5 *(Interval word) Given a poset $P = (V_P, <_P)$, an* **interval word** *IW of P is a word over $V_P \times \{S, F\}$ such that for all events v_1, v_2:*

1. *Each letter from the alphabet $V_P \times \{S, F\}$ appears exactly once in IW.*
2. *$v_1 <_P v_2$ implies that $\langle v_1, F\rangle$ appears before $\langle v_2, S\rangle$ in IW.*
3. *For each event v, $\langle v, S\rangle$ appears before $\langle v, F\rangle$.*

An interval word of a pomset is an interval word of the underlying poset.

An interval word is called *characteristic* if 2 holds for both directions. More formally:

Definition 2.6 *Given a poset $P = (V_P, <_P)$, a* **characteristic interval word** *IW of P is a word over $V_P \times \{S, F\}$ such that for all events v_1, v_2:*

1. *Each letter from the alphabet $V_P \times \{S, F\}$ appears exactly once in IW.*
2. *$v_1 <_P v_2$ iff $\langle v_1, F\rangle$ appears before $\langle v_2, S\rangle$ in IW.*
3. *For each event v, a letter $\langle v, S\rangle$ appears before $\langle v, F\rangle$.*

It is clear that P has a characteristic interval word iff it is an interval poset. A characteristic interval word of a pomset is a characteristic interval word of the underlying poset.

Definition 2.7 *For a pomset P and a label l, we define $label\_width(P, l)$ as the maximal number of multually concurrent events in P labeled by l. We define $max\_width(P)$ as the maximum $label\_width(P, l)$ for all $l \in \Sigma$. For a pomset language PL we define $max\_width(PL)$ to be $max\{max\_width(P) | P \in PL\}$.*

2.2 Split Equivalence

Definition 2.8 *(language environment) Let Σ be a set of labels. A* language environment *α is a function that assigns a language $\alpha(A)$ for every label $A \in \Sigma$.*

Definition 2.9 *(parsing) Given a pomset P, a word s and a language-environment α, a parsing of s with respect to P and α is a function $h : \{1 \dots length(s)\} \to V_P$ such that:*

1. *$v_1 <_P v_2$ implies that $h^{-1}(v_1)$ precedes $h^{-1}(v_2)$ (i.e. the biggest index in $h^{-1}(v_1)$ is smaller than the smallest index of $h^{-1}(v_2)$)*
2. *$h^{-1}(v)$ is a set of indices $i_1 < i_2 < \dots < i_n$ for some n, and $s[i_1] \dots s[i_n] \in \alpha(lab_P(v))$*

Definition 2.10 *Let P be a pomset. $P \bullet \alpha$ is defined as the set of all strings s such that there exists a parsing of s with respect to P and α. For a pomset-language PL we define $PL \bullet \alpha$ as the union of $P \bullet \alpha$ for all $P \in PL$.*

Next, we consider a special case of language-environment.

Definition 2.11 *The language environment $split_n$ assigns to every label A the single-string language $\{ \langle A, 1\rangle \ \langle A, 2\rangle \ .. \ \langle A, n\rangle \}$.*

In this paper we almost always consider the $split_n$ language environment. Therefore, when we use the term *parsing* without stating the environment, we refer to $split_n$, where n will be clear from the context. For the sake of completeness, we now present a direct definition of parsing with respect to $split_n$. The reader will observe that this definition agrees with definitions 2.9 and 2.11.

Definition 2.12 *(parsing with respect to $split_n$) Given a pomset P and a word s over $\Sigma \times Nat$, a parsing of s with respect to P and $split_n$ is a function $h : \{1 \dots length(s)\} \rightarrow V_P$ such that:*

1. *$v_1 <_P v_2$ implies that $h^{-1}(v_1)$ precedes $h^{-1}(v_2)$ (i.e. the biggest index in $h^{-1}(v_1)$ is smaller than the smallest index in $h^{-1}(v_2)$)*
2. *$h^{-1}(v)$ is a set of indices $i_1 < i_2 < \dots < i_n$, and $s[i_j] = \langle lab_P(v), j\rangle$ for all $j = 1 \dots n$*

Definition 2.13 *The pomset P' is called an* **augmentation** *of a pomset P if P' has the same set of events as P and the same labeling functions, and $<_{P'} \supseteq <_P$.*

We denote by $Aug(P)$ the set of all augmentations of P.

The following lemma is immediate.

Lemma 2.14 *If $P \in Aug(Q)$, then $\forall m. P \bullet split_m \subseteq Q \bullet split_m$*

The last lemma is valid for every language environment. Moreover, there exists a stronger result:

Theorem 2.15 *[6, 7] Let PL_1 and PL_2 be two pomset languages. Then $PL_1 \bullet \alpha = PL_2 \bullet \alpha$ for every language-environment α iff for every interval pomset P we have $P \in Aug(PL_1) \leftrightarrow P \in Aug(PL_2)$.*

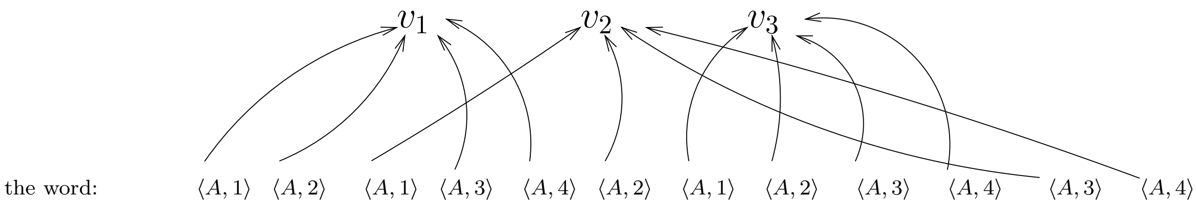


Figure 4: A parsing of a word with respect to a pomset.

Similar to the definition of a characteristic interval word, we define a characteristic parsing.

Definition 2.16 *(characteristic parsing) A parsing h of a word s with respect to a pomset P is called* **characteristic** *when $v_1 <_P v_2$ iff $h^{-1}(v_1)$ precedes $h^{-1}(v_2)$.*

We now define a new equivalence for pomset languages.

Definition 2.17 *(split equivalence) Two pomset languages PL_1 and PL_2 are* split-n equivalent *if $PL_1 \bullet split_n = PL_2 \bullet split_n$.*

Definition 2.18 *Two pomset languages PL_1 and PL_2 are* split-equivalent *if they are* split-n equivalent *for every n.*

It is easy to see that if PL_1 and PL_2 are split-equivalent than for every α that assigns strings to the labels of PL_1 and PL_2, the languages $PL_1 \bullet \alpha$ and $PL_2 \bullet \alpha$ coincide.

Similarly (see [12]), if PL_1 and PL_2 are split-n equivalent and α assigns to every label a string of length at most n, then also $PL_1 \bullet \alpha = PL_2 \bullet \alpha$.

2.3 Shuffle-regular expressions and Petri nets

Shuffle regular expressions are defined by the following grammar:

$E ::= \ X \mid c \mid E + E \mid E;E \mid E||E \mid E^*$, where X ranges over an alphabet of variable symbols and c ranges over an alphabet of constant symbols.

We say that E is a variable free expression if it contains no variables. We say that E is a constant free expression if it contains no constants.

A string language is a set of strings. The operations sum, concatenation, iteration and shuffle are defined in a standard way on string languages. We recall that a string w belongs to the shuffle of languages L_1 and L_2 if there exist strings $u_1u_2\dots u_k$ and $w_1w_2\dots w_k$ such that
$w = w_1u_1w_2u_2\dots w_ku_k$ and $w_1w_2\dots w_k \in L_1$ and $u_1u_2\dots u_k \in L_2$.

For an expression E the string language $L(E)$ is assigned in a standard way. We say that $L(E)$ is the language defined by the expression E.

Definition 2.19 *The shuffle width $sw(E)$ of an expression E is defined inductively by:*

- $sw(X) = sw(c) = 1$
- $sw(E_1 + E_2) = sw(E_1; E_2) = max(sw(E_1), sw(E_2))$
- $sw(E_1 \parallel E_2) = sw(E_1) + sw(E_2) + 1$
- $sw(E^*) = sw(E)$

Definition 2.20 *Given an expression E with variables $X_1 \dots X_m$, we define $string_n(E)$ as the variable-free expression obtained from E by instantiating every variable X_i by the string $x_i^1 \dots x_i^n$.*

Definition 2.21 *Two expressions E_1 and E_2 are said to be* string-n-equivalent *(notation $E_1 \sim_{str}^n E_2$) if $string_n(E_1)$ and $string_n(E_2)$ define the same language. E_1 and E_2 are said to be* string-equivalent *(notation $E_1 \sim_{str} E_2$)if they are string-n-equivalent for all $n \in Nat$.*

It can be shown (see [12]) that $E_1 \sim_{str}^n E_2$ iff $E_1 = E_2$ under every instantiation of the variables by strings with length $\leq n$.

We observe that string-n-equivalence is decidable.

Lemma 2.22 *For every fixed n it is decidable whether two expressions are string-n-equivalent.*

Proof: Given two expressions, apply $string_n$ on both of them (i.e. instantiate every variable X_i by the string $x_1^i \dots x_n^i$) and check language equivalence. Checking language equivalence is comparing between finite automata. $\square$

Following, we have the connection between expressions and pomsets. For simplicity we will deal only with constant-free expressions. In section 11.1 we comment about the extension of the result for expressions with constants.

Theorem 2.23 *(see [6]) There exists a function that assigns to each constant-free shuffle-regular expression E a pomset language $PL(E)$ such that:*

- $\forall E.wid(PL(E)) = sw(E)$
- $\forall n.PL(E) \bullet split_n = L(string_n(E))$

Figure 2 shows several examples for assigning expressions with pomset-languages.
A similar theorem exists for C/E Petri nets (see [13, 8, 10]). We now give this theorem, preceded by some basic definitions.

Definition 2.24 *For a Petri net N, we define $max\_width(N)$ to be the maximum number of mutualy concurrent transitions labeled by the same label in N.*

Figure 5 shows a transition in a Petri net. In the figure, boxes represent the places of the net, while rectangles represent its transitions. As seen in the figure, the transition t has two sets of places - $A = preconditions(t)$ and $B = postconditions(t)$.
Now let n be a natural number. As demonstrated in figure 6 (for the case $n = 4$), *Splitting in n* of the transition t is the operation that replaces t by n transitions $t_1, t_2 \dots t_n$, and adds an additional $n-1$ places $p_1, p_2 \dots p_{n-1}$. The relations between the new places and transitions are defined as follows:

- $preconditions(t_1) = A$
- $preconditions(t_i) = \{p_i\}$ for each $i = 2 \dots n$
- $postconditoins(t_i) = \{p_{i+1}\}$ for each $i = 1 \dots n-1$
- $postconditions(t_n) = B$

Given a Petri net N, we denote by $split_n(N)$ the Petri net obtained by splitting each transition of N to n transitions.

Definition 2.25 *Let N_1 and N_2 be Petri nets. We say that N_1 is split-n-equivalent to N_2 if $split_n(N_1)$ and $split_n(N_2)$ define the same language. We say that they are split-equivalent if they are split-n-equivalent for every n.*

Theorem 2.26 *(see [13]) There exists a function that assigns to each C/E Petri net N a pomset language $PL(N)$ such that:*

- $\forall N.max\_width(PL(E)) = max\_width(N)$.
- $\forall n.PL(N) \bullet split_n = L(split_n(N))$.

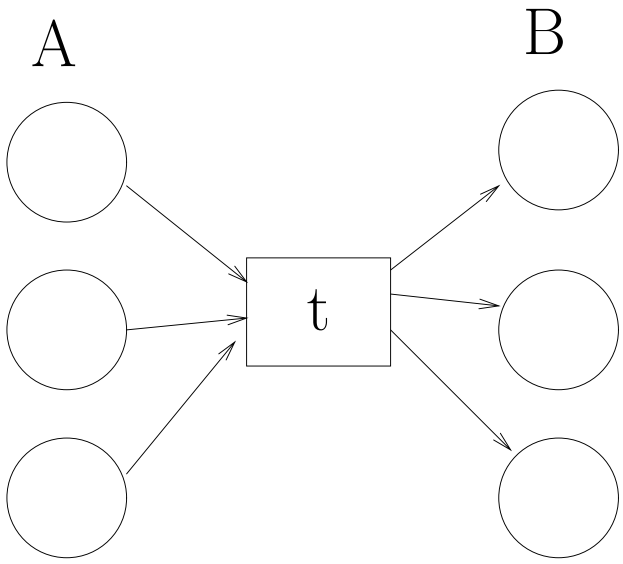


Figure 5: A transition in a Petri net

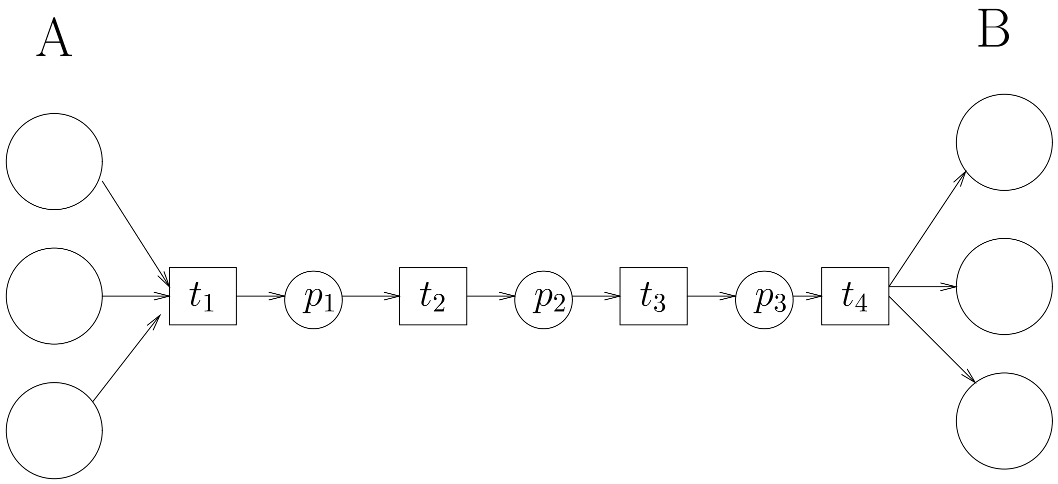


Figure 6: A splitting of a transition

3 Main results

Theorem 3.1 *Let PL_1 and PL_2 be two pomset languages. Suppose that for every $k \leq 1 + max\_width(PL_1)$ the inclusion $PL_1 \bullet split_k \subseteq PL_2 \bullet split_k$ holds, then $\forall m. PL_1 \bullet split_m \subseteq PL_2 \bullet split_m$.*
In particular if $\forall k \leq 1 + max(max\_width(PL_1), max\_width(PL_2))$ we have that PL_1 and PL_2 are split-k-equivalent, then PL_1 and PL_2 are split-equivalent.

Corollary 3.2 *1. It is decidable whether two shuffle-regular expressions are string-equivalent.*

2. Let E_1 and E_2 be two shuffle-regular expressions. If for every $k \leq max(sw(E_1), sw(E_2)) + 1$, the expressions are string-k equivalent, then they are string-equivalent.

Proof: Immediate from theorems 3.1, 2.23 and 2.22. □

Corollary 3.3 *1. It is decidable whether two finite Petri nets are split-equivalent.*

2. Let N_1 and N_2 be two finite Petri nets. If $\forall k \leq max(wid(N_1), wid(N_2)) + 1$, the nets are split-k-equivalent, then they are split-equivalent.

Proof: Immediate from theorems 3.1 and 2.26. □

Theorem 3.1 relies on the following theorem.

Theorem 3.4 *Given an interval pomset P, there exists a word s over $\Sigma \times Nat$ such that:*

1. *$s \in P \bullet split_{max\_width(P)+1}$*
2. *For any interval pomset Q if $s \in Q \bullet split_{max\_width(P)+1}$ then $\forall m. P \bullet split_m \subseteq Q \bullet split_m$.*

3.1 Proof of Theorem 3.1

Let Int be the set of interval pomsets. Given PL_1 and PL_2, we define $PL'_1 = Aug(PL_1) \cap Int$ and $PL'_2 = Aug(PL_2) \cap Int$. Observe that:

1. PL'_i has the same width as PL_i (for $i = 1, 2$).
2. $PL'_1 \bullet split_n = PL_1 \bullet split_n$ (by Theorem 2.15).

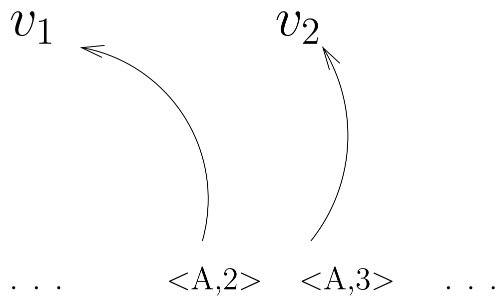


Figure 7: A change of a parsing

3. $PL'_2 \bullet split_n = PL_2 \bullet split_n$ (by Theorem 2.15).
4. PL'_1 and PL'_2 contain only interval pomsets. Moreover, PL'_1 and PL'_2 contain all the interval pomsets of $Aug(PL_1)$ and $Aug(PL_2)$, respectively.

Assume that for some m, some word w is in $PL_1 \bullet split_m$. By (2) we have $w \in PL'_1 \bullet split_m$. Therefore, for some pomset $P \in PL'_1$, $w \in P \bullet split_m$. By (4) P is an interval pomset. By Theorem 3.4 there exists a word $s \in P \bullet split_k$ for $k = max\_width(P) + 1 \leq max\_width(PL'_1) + 1 = max\_width(PL_1) + 1$ (the last equality is by (1)) with the properties specified in the theorem. By hypothesis (that $PL_1 \bullet split_k \subseteq PL_2 \bullet split_k$) and by (4) there exists an interval pomset $Q \in PL'_2$ such that $s \in Q \bullet split_k$. By Theorem 3.4 we have that $\forall m. P \bullet split_m \subseteq Q \bullet split_m$, so $w \in Q \bullet split_m$ and $w \in PL'_2 \bullet split_m$. By (3), $w \in PL_2 \bullet split_m$.

3.2 Proof of Theorem 3.4

Our proof of Theorem 3.4 will rely on the three lemmas stated below.

Definition 3.5 *(change) Given a pomset P over an alphabet Σ, a word s over $\Sigma \times Nat$ and a parsing h of s with respect to P, it is said that there is a* **change** *in h at place i, if:*

1. $s[i] = \langle A, k \rangle$ *and* $s[i-1] = \langle A, k-1 \rangle$ *for some* $A \in \Sigma$
2. $h[i] \neq h[i-1]$

Figure 7 illustrates a change of a parsing.

Definition 3.6 *A parsing is* **change-free** *if it contains no changes.*

Notations: Given a parsing h, let $change\_number(h)$ be the number of changes in h. Given a pomset P and a word s over $\Sigma \times Nat$, let $min\_change\_number(P, s) = min\{change\_number(h) | h$ is a parsing of s with respect to $P\}$

For the following lemmas we need the notion of an **n-distinctive** word, which will be defined later (see Definition 4.4).

Lemma 3.7 *Let P be an interval pomset, Q a pomset. Let s be an n-distinctive word, where $n = max\_width(P) + 1$. Let h and h' be change-free parsings of s with respect to P and Q respectively, such that h is characteristic, then $P \in Aug(Q)$*

Proof: see section 6

Lemma 3.8 *Given an interval pomset Q, an n-distinctive word s over $\Sigma \times Nat$, where $n \in Nat$, and a parsing h of s with respect to Q, such that $change\_number(h) = min\_change\_number(Q, s) > 0$, then there exists a function h' and a pomset Q' such that:*

1. *h' is a parsing of s with respect to Q'*
2. *$\forall m.Q' \bullet split_m \subseteq Q \bullet split_m$*
3. *$change\_number(h') < change\_number(h)$*

Proof: see section 8.

Lemma 3.9 *Given an interval pomset P, there exist an n-distinctive word s over $\Sigma \times Nat$, where $n = max\_width(P) + 1$, and a change-free characteristic parsing h of s with respect to P.*

Proof: see section 5.

We now prove Theorem 3.4 using these lemmas.
Let s be the word from Lemma 3.9. Lemma 3.9 (and Definition 4.4) immediately implies that $s \in P \bullet split_{max\_width(P)+1}$, so the first part of the theorem is proved. For the second part, assume that there is a pomset Q such that $s \in Q \bullet split_{max\_width(P)+1}$. We prove the theorem by induction on $min\_change\_number(Q, s)$:
Induction base: if $min\_change\_number(Q, s) = 0$ then there exists some parsing h' such that $change\_number(h') = 0$. Therefore h' is a change-free parsing. By Lemma 3.7 we have that $P \in Aug(Q)$ and by Lemma 2.14 $\forall m.P \bullet split_m \subseteq Q \bullet split_m$.
Induction step: Assume that $min\_change\_number(Q, s) > 0$, then there exists a parsing h_1 such that $change\_number(h_1) = min\_change\_number(Q, s) > 0$. By Lemma 3.8 there exists Q' and a parsing h' of s with respect to Q', such that $change\_number(h') < change\_number(h)$. By the induction hypothesis $\forall m.P \bullet split_m \subseteq Q' \bullet split_m$. Lemma 3.8 also gives $\forall m.Q' \bullet split_m \subseteq Q \bullet split_m$, so we have $\forall m.P \bullet split_m \subseteq Q \bullet split_m$.

4 Distinctive word

In this section we define a word which will satisfy the conditions of Theorem 3.4, a word that will distinguish between different pomsets.

Definition 4.1 *(capacity) For any $X \in \Sigma$, indices $i, k > 0$, and a word s over $\Sigma \times Nat$, we define $capacity(X, s, i, k)$ to be the number of occurrences of the letter $\langle X, k \rangle$ in $s[1 \ldots i]$, minus the number of occurrences of the letter $\langle X, k+1 \rangle$ in $s[1 \ldots i]$.*

Definition 4.2 *(well-formedness) A string s is called* well-formed *if there exists an n such that $s \in t_1 \parallel t_2 \parallel \ldots \parallel t_k$ where each $t_i = \langle A, 1 \rangle\ \langle A, 2 \rangle\ ..\ \langle A, n \rangle$ for some A*

The proof of the following lemma is clear.

Lemma 4.3 *Given a pomset P and a word s, if $s \in P \bullet split_n$ (for some n), then s is well-formed.*

Definition 4.4 *(distinctive word) A word s over $\Sigma \times \{1, \ldots n\}$ is* n-distinctive *if the following conditions hold:*

- *s is well formed*
- *s can be partitioned into k subsequent substrings $s[i_j \ldots i_{j+1} - 1], j = 1 \ldots k$ which belong to one of the following types:*
  1. **Decreasing** *substring $s_j = s[i_j \ldots i_{j+1} - 1]$, in which:*
     - *(a) $s_j = \langle A, m \rangle \langle A, m-1 \rangle ... \langle A, 1 \rangle$ for some $m < n$*
     - *(b) $capacity(A, s, i, k) \leq 1$ for all $A \in \Sigma$, $0 < k < n$, $i_j \leq i < i_{j+1}$.*
  2. **Increasing** *substring $s_j = s[i_j \ldots i_{j+1} - 1]$, in which:*
     - *(a) $s_j = \langle A, m \rangle \langle A, m+1 \rangle ... \langle A, n \rangle$ for some $m > 1$*
     - *(b) $capacity(A, s, i, k) \leq 2$ for all $A \in \Sigma$, $0 < k < n$, $i_j \leq i < i_{j+1}$.*

Whenever n is clear from the context, we use *distinctive* for n-distinctive.

Figure 8 shows an example for a 4-distinctive word. In the figure, the parts are marked with D (decreasing) and I (increasing).

Lemma 4.5 *A distinctive word s can be partitioned in the above manner in only one way.*

Proof: Clear from conditions 1(a) and 2(a). □

$$\underbrace{\langle A,1\rangle}_{D}\,\underbrace{\langle A,2\rangle\langle A,1\rangle}_{D}\,\underbrace{\langle A,3\rangle\langle A,2\rangle\langle A,1\rangle}_{D}\,\underbrace{\langle A,4\rangle}_{I}\,\underbrace{\langle A,3\rangle\langle A,2\rangle\langle A,1\rangle}_{D}\,\underbrace{\langle A,3\rangle\langle A,4\rangle}_{I}\,\underbrace{\langle A,2\rangle\langle A,1\rangle}_{D}\,\underbrace{\langle A,4\rangle}_{I}\,\underbrace{\langle A,3\rangle\langle A,4\rangle}_{I}\,\underbrace{\langle A,2\rangle\langle A,3\rangle\langle A,4\rangle}_{I}$$

Figure 8: A 4-distinctive word.

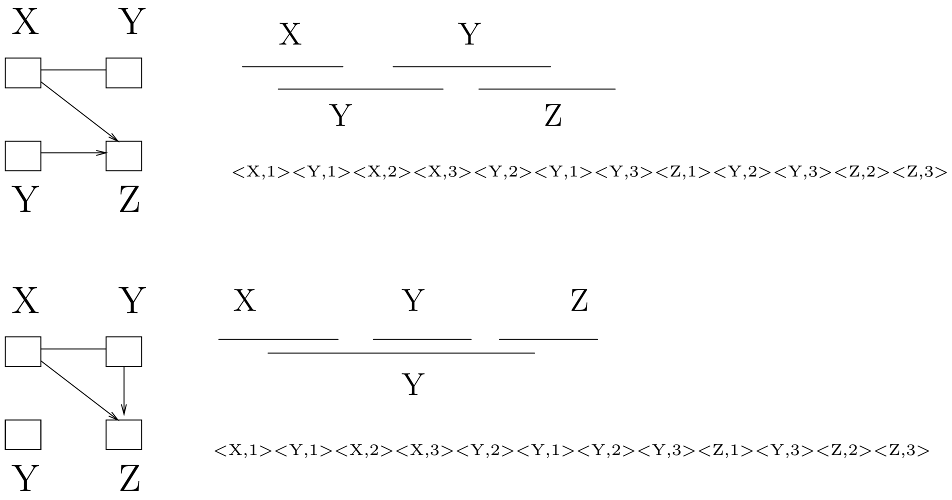


Figure 9: word construction

5 The existence of a distinctive word

Recall that Lemma 3.9 states that, for every interval pomset P, there exists a $(max\_width(P)+1)$-distinctive word s over $\Sigma \times Nat$ and a change-free characteristic parsing h of s with respect to P.

We describe an algorithm that takes an interval pomset P and produces the desired distinctive word s. The algorithm also produces a parsing of s with respect to P.

5.1 Input and output of the algorithm

To produce a word, we take the characteristic interval word of the input pomset P. Such exists, because it is an interval pomset.

During the algorithm we write out a stream of triples $\langle A, v, i\rangle$, where A is a label, v is an event of the pomset and i is an index. The following lemma describes the use of this stream.

Lemma 5.1 *The stream $\langle X_1, v_1, i_1\rangle, \langle X_2, v_2, i_2\rangle, \dots \langle X_m, v_m, i_m\rangle$ produced by the algorithm satisfies the following conditions:*

1. $m = (max\_width(P) + 1) \times (size(V_P))$
2. *The string* $w = \langle X_1, i_1 \rangle \; \langle X_2, i_2 \rangle \; ... \; \langle X_m, i_m \rangle$ *is a distinctive word.*
3. *The function* $f(i) = v_i$ *is a change-free characteristic parsing of* w *with respect to* P*.*

Figure 9 shows two interval pomsets of width 2, their intervals and the 3-distinctive words that are constructed by the algorithm.

5.2 Activation records

During the run of the algorithm we maintain an *activation record* for each label $A \in \Sigma$. This activation record is a table of size $max\_width(P)$, in which events can reside.
An event can be in 3 status: fresh, active and completed.

- At the beginning of the algorithm, all events are fresh. A fresh event is not in an activation record.
- Every event becomes active at some stage of the algorithm and enters the activation record of its label. It starts in entry (cell) 1 of the record and moves increasingly to stage $m - 1$.
- An event becomes complete after being active, leaves the activation record, and will not change status anymore. The algorithm will no longer involve this event.

5.3 The algorithm

TheAlgorithm
input: a characteristic interval word W of a pomset P (the length of W is $size(V_P) * 2$)
output: a word $w \in P \bullet split_{max\_width(P)+1}$ and a parsing h of w with respect to P
begin
for i = 1 to length(W)
    if $W[i] = \langle v, S \rangle$ then
        // start of new event: produce "decreasing part"
        Let X be the label of v
        consider the activation record of the label X
        find the smallest empty place k in the activation record.
        for j from $k - 1$ to 1
            for all events v' in entry j of the table
                move event v' to cell $j + 1$
                and write out $\langle X, v', j + 1 \rangle$
        //now the lowest cell is empty and ready to accept new event)

move event v to cell 1, change status of v to active.
write out $\langle X, v, 1\rangle$
else if $W[i] = \langle v, F\rangle$ then
// end of event : produce "increasing part")
Let X be the label of v
consider the activation record of the label X.
Say v in cell k of this record
// move the event up until it gets out of the activation record)
for j from $k+1$ to $max\_width(P)+1$
move v to cell j and write out $\langle X, v, j\rangle$.
// last move in this loop takes the event out of the activation record
change status of v to complete.
end-if
end

Figure 10 show a complete run of the algorithm, with the input pomset, the intervals of the pomset, the output word and the status of the activation records during the run.

5.4 Correctness of the algorithm

We observe the following invariants during the run of the algorithm:

1. Every active event occupies at most one entry of an activation record at any time.
2. During the decreasing part of the algorithm every entry contains at most one event, and a substring of the form $\langle X, v_1, m\rangle, \langle X, v_2, m-1\rangle, \ldots \langle X, v_m, 1\rangle$ is produced.
3. During the increasing part for an event v, every entry contains at most one event distinct from v, and a substring of the form $\langle X, v, m\rangle, \langle X, v, m+1\rangle, \ldots \langle X, v, max\_width(P)+1\rangle$ is produced.
4. If $s[i] = \langle v, S\rangle$, then v becomes active at iteration i.
5. If $s[i] = \langle v, F\rangle$, then v becomes complete at iteration i.
6. Assume that i letters have been written to the output stream, then $capacity(A, w, i, k)$ is equal to the number of events in entry k of the activation record of A (where w is the word produced by the algorithm).

The correctness of the algorithm includes two assertions:

- Successful termination.

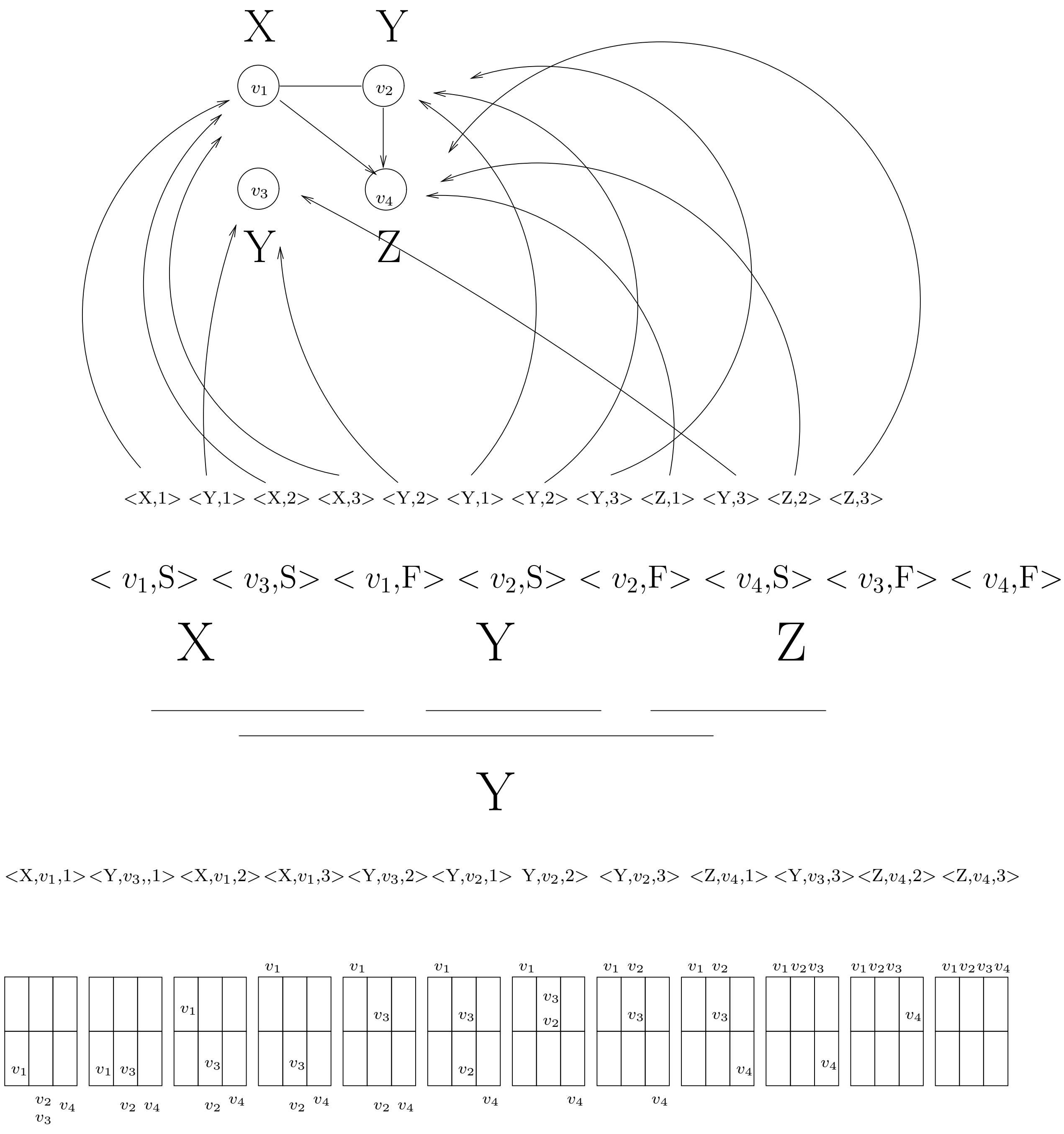


Figure 10: A run of the algorithm

- Correctness of the output (Lemma 5.1).

It is clear that the algorithm terminates. The only case in which the algorithm cannot terminate successfully is if there is no free cell in an activation record of a label A. Note that by invariants 4 and 5, a full record implies $max\_width(P)$ events that have started in W and not yet finished. In addition to the event being handled (marked as v in the algorithm), that makes $max\_width(P)+1$ events that have started but not finished. Since W is a characteristic interval word, this implies $max\_width(P)+1$ concurrent events, which is impossible.

Part 2 of Lemma 5.1 is straightforward, in view of invariants 6, 2 and 3. For part 3 of Lemma 5.1, we first prove that $f(i) = v_i$ is a parsing. Assume that some $v_1 <_P v_2$. Since W is characteristic, we have that $\langle v_1, F\rangle$ appears before $\langle v_2, S\rangle$. By invariants 4 and 5, v_1 is complete before v_2 becomes active. Thus, any output letter that contains v_1 will precede a letter with v_2. These implications are valid to both directions, so we have that the parsing is characteristic.

Next we show that the parsing is change-free. This is straightforward because a change can be only in an increasing part, but such part has the form $\langle X, v, m\rangle, \langle X, v, m+1\rangle, \ldots \langle X, v, max\_width(P)+1\rangle$, hence it corresponds to one event.

6 Proof of Lemma 3.7

Definition 6.1 *Given pomsets P and Q and a word s over $\Sigma \times Nat$, Let h and h' be parsings of s with respect to P and Q respectively. We say that h and h' are isomorphic (notation $h \sim h'$) if there exists a bijective function $I : V_P \to V_Q$ such that for each $v \in V_P$, $h^{-1}(v) = h'^{-1}(I(v))$.*

Lemma 6.2 *Let P, Q be two pomsets. Let s be a distinctive word. Let h and h' be change-free parsings of s with respect to P and Q respectively. Then $h \sim h'$.*

Proof: We build a Relation $R \subseteq V_P \times V_Q$. For each $v \in V_P, v' \in V_Q$ we define:
vRv' iff there is i such that

1. $h(i) = v$
2. $h'(i) = v'$
3. $s[i] = \langle A, 1\rangle$ for some label A

Since for every $v \in V_P$ (respectively $v \in V_Q$) there exists a unique $i \in h^{-1}(v)$ (respectively $h'^{-1}(v)$) such that $s[i] =\langle A, 1\rangle$, it follows that R is the graph of label-preserving bijection between P and Q. Let $I : V_P \to V_Q$ be that bijection. We argue that the function I is an isomorphism between the parsings h and h'.

We have to show that for each $v \in V_P, v' \in V_Q$, if vRv' then $h^{-1}(v) = h'^{-1}(v')$.
Assume that this is not the case. Let j be the smallest index such that $j \in h^{-1}(v) \triangle h'^{-1}(v')$ (where $\triangle$ denotes symmetrical difference). We can assume w.l.g. that $j \in h^{-1}(v) \setminus h'^{-1}(v')$.
Assume that $s[j]=\langle A, m\rangle$, for some label A. By the definitions of R and I, m cannot be 1.
$h^{-1}(v) \cap h'^{-1}(v')$ contains $m-1$ indices $j_1 < j_2 \ldots < j_{m-1}$ such that $h(j_i) = v, h'(j_i) = v', s[j_i] = \langle A, i\rangle$ for all $i = 1 \ldots m-1$.

We proceed by the following cases:

1. $j = j_{m-1} + 1$. By hypothesis, $h'(j) \neq v'$ and $h'(j_{m-1}) = v'$, therefore $h'(j-1) \neq h'(j)$. But this implies a change in h' at place j, and this is a contradiction to the fact that h' is change-free.
2. Otherwise. Let v'' be $h'(j)$. By hypothesis $v'' \neq v'$. Note that $s[j] =\langle A, m\rangle$. Thus, the event v'' appears $m-1$ times on $h'(1 \ldots j-1)$. The same holds for v', and this implies *capacity*$(A, s, j-1, m-1) \geq 2$ (by definition of capacity - Definition 4.1).

Since s is distinctive we must have that $j-1$ is in an increasing part of the word s, so $s[j-1] = \langle A, m-1\rangle$ and $s[j] = \langle A, m\rangle$. $h(j-1) \neq v$ because if $h(j-1) = v$ then $j-1 \in h^{-1}(v)$ and moreover $j-1 = j_{m-1}$, but this was the previous case. So $h(j-1) \neq v = h(j)$. But this implies a change in h at place j. Contradiction.

□

We are now ready to prove Lemma 3.7. Let P be an interval pomset, Q a pomset. Let s be a distinctive word. Let h and h' be change-free parsings of s with respect to P and Q respectively, such that h is characteristic. In these conditions, we have to show that $P \in Aug(Q)$.
By Lemma 6.2 we have $h \sim h'$, so there exists a function $I : V_Q \rightarrow V_P$ which is an isomorphism between the parsings h and h' . We show that this function respects the following conditions:

- I is label preserving. That is, for every $v \in V_P$ we have $lab_P(v) = lab_Q(I(v))$.
- For every two events v_1 and v_2, if $v_1 <_Q v_2$ then $I(v_1) <_P I(v_2)$

Thus showing that I is an isomorphism between P and an augmentation of Q. Since we treat isomorphic pomsets as equal, this is enough.
The label-preserving is straightforward. For the second condition, assume $v_1 <_Q v_2$. h' is a parsing, so we have that $h'^{-1}(v_1)$ precedes $h'^{-1}(v_2)$. Because $h \sim h'$, we have that there are $v_3, v_4 \in V_P$ (namely $v_3 = I(v_1)$, $v_4 = I(v_2)$) such that $h^{-1}(v_3) = h'^{-1}(v_1)$ precedes $h^{-1}(v_4) = h'^{-1}(v_1)$. Since h is characteristic, we have $v_3 = I(v_1) <_P v_4 = I(v_2)$.

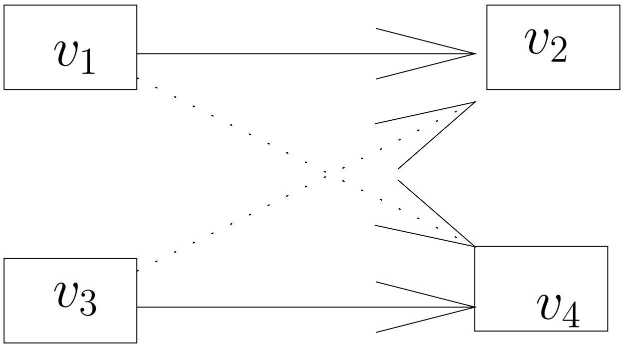


Figure 11: N-shape

7 The swap operation on pomsets

Our goal now is to progress towards a proof of Lemma 3.8. For the proof of this lemma we resort to the **swap** operation, defined in [13].

Definition 7.1 *(simple swap) Given an interval pomset P and two concurrent events v_1 and v_2 such that $lab_P(v_1) = lab_P(v_2)$, we define a new pomset $P' = simple\_swap(P, v_1, v_2) = (V_P, <_{P'}, lab_P)$, where:*

$$v <_{P'} v' iff \begin{cases} v_2 <_P v' & \text{if } v = v_1 \\ v_1 <_P v' & \text{if } v = v_2 \\ v <_P v' & \text{otherwise} \end{cases}$$

This operation can be viewed as described in figure 12: it replaces the successors of v_1 and v_2, the two swapped events.

Remark 7.2 *(the N-shape property of interval pomsets) It was shown in [14], that a pomset P is an interval pomset iff for every $v_1, v_2, v_3, v_4 \in V_P$ we have: if $v_1 <_P v_2$ and $v_3 <_P v_4$, then $v_1 <_P v_4$ or $v_3 <_P v_2$. Thus, whenever we have the "parallel arrows" drawn in figure 11, we must also have one of the 'diagonals' indicated by the dotted lines. The reader will observe that because of this reason, the result of application of simple_swap operation to an interval pomset P is indeed a pomset (i.e. $<_{P'}$ is a partial order). Moreover this pomset is an interval pomset.*

Definition 7.3 *Given an interval pomset P and two concurrent events v_1 and v_2, we say that v_1 and v_2 are in* **swap configuration** *if there exist v_0 and v_3 such that $v_0 <_P v_1$, v_0 co_P v_2, $v_2 <_P v_3$ and v_1 co_P v_3, or the symmetric case (i.e. by exchanging v and v' in all four conditions).*

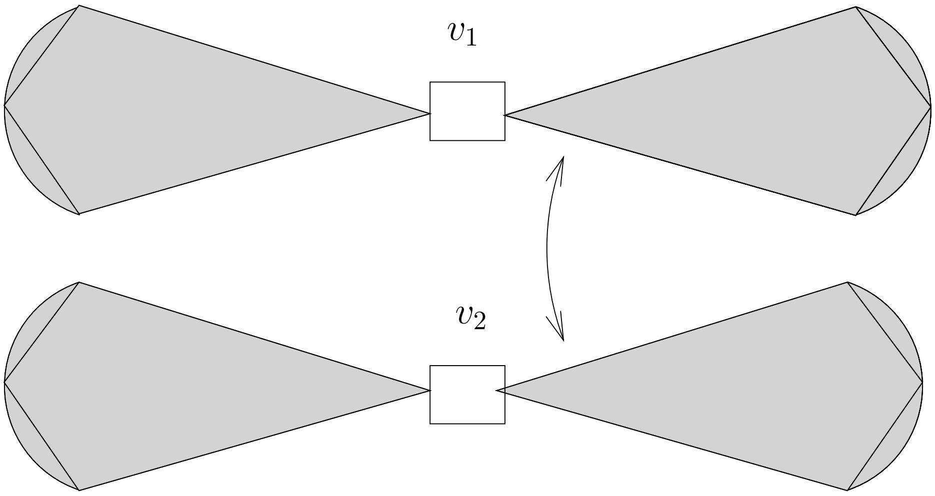


Figure 12: A simple-swap operation on a pomset

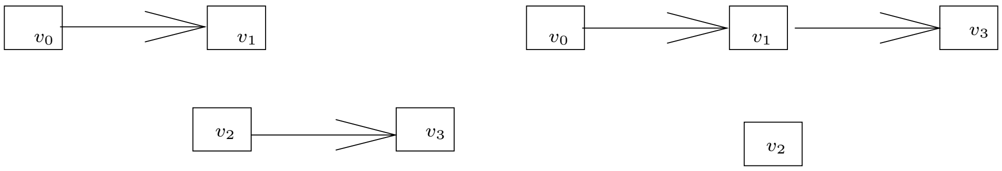


Figure 13: swap and anti-swap configuration

Definition 7.4 *Given an interval pomset P and two concurrent events v_1 and v_2, we say that v_1 and v_2 are in* **anti-swap configuration** *if there exist v_0 and v_3 such that $v_0 <_P v_1$, $v_0 \; co_P \; v_2$, $v_1 <_P v_3$ and $v_2 \; co_P \; v_3$, or the symmetric case.*

Figure 13 shows pomsets in swap and anti-swap configurations. Note that two events can be in swap or in anti-swap configuration, or in none of them, but not in both, since P is an interval pomset. This is because the N-shape propery of interval pomsets.
The reader will observe that this property of an interval pomset does not allow two events to be both in swap and anti-swap configuration.

Vogler defines a swap operation on two concurrent events that are in swap-configuration. Indeed, as we will further see, the swap operation of two events is affective only if they are in swap (or anti-swap) configuration.

Definition 7.5 *(swap, Vogler [13]) Given pomsets P and Q, we say that $P \in swap(Q)$*

if $P = simple\_swap(Q, v_1, v_2)$ for some $v_1, v_2 \in V_Q$ such that v_1 and v_2 are in swap configuration. We define $swap^*(Q)$ as the iterative closure of $swap(Q)$.

Observe that $swap^*(Q)$ is well defined because $swap(Q)$ is an interval pomset if Q is an interval pomset.
Figure 14 gives an example of the swap operation.

Lemma 7.6 *[13] Let P and Q be interval pomsets. If $Q \in swap^*(P)$, then $\forall m. Q \bullet split_m \subseteq P \bullet split_m$*

The following example gives some intuition to lemma 7.6 and to the relations between *swap* and $split_n$ in general. These relations are fully described in Vogler's paper ([13]).
In figure 15 we can see two interval pomsets P_a and P_b such that $P_b \in swap(P_a)$. Near the pomset, we can see their corresponding interval words I_a and I_b. Consider now the following two operations:

1. Let v be the i'th event labeled by X. Replace $\langle v, S\rangle$ by $\langle X_i, 1\rangle$, and $\langle v, F\rangle$ by $\langle X_i, 2\rangle$
   In this case we obtain from I_a and I_b, respectivly, the strings
   $\langle X_1, 1\rangle\langle Y_1, 1\rangle\langle X_1, 2\rangle\langle Y_2, 1\rangle\langle Y_1, 2\rangle\langle Z_1, 1\rangle\langle Y_2, 2\rangle\langle Z_1, 2\rangle$
   and
   $\langle X_1, 1\rangle\langle Y_1, 1\rangle\langle X_1, 2\rangle\langle Y_2, 1\rangle\langle Y_2, 2\rangle\langle Z_1, 1\rangle\langle Y_1, 2\rangle\langle Z_1, 2\rangle$.

2. Let v an event labeled by X. Replace $\langle v, S\rangle$ by $\langle X, 1\rangle$, and $\langle v, F\rangle$ by $\langle X, 2\rangle$
   In this case we obtain from both I_a and I_b the same string
   $\langle X, 1\rangle\langle Y, 1\rangle\langle X, 2\rangle\langle Y, 1\rangle\langle Y, 2\rangle\langle Z, 1\rangle\langle Y, 2\rangle\langle Z, 2\rangle$.

Note, that the strings we obtain by the first replacement contain all the information to reconstruct the interval words (and the pomsets). This replacement is similar to the one done in Theorem 1.1. From the second replacement, however, we obtain the same string from which, of course, the pomsets cannot be reconstructed.
Replacement (2) is actualy $split_2$ application. Hence, the string which is obtained from replacement (2) belongs to $P_a \bullet split_2$, and also to $P_b \bullet split_2$. This example gives some intuition to lemma 7.6

Lemma 7.7 *Assume that $Q' = simple\_swap(Q, v_1, v_2)$, then:*

- *If v_1 and v_2 are in swap configuration (in Q), then $Q' \in swap(Q)$.*
- *If v_1 and v_2 are in anti-swap configuration, then $Q \in swap(Q')$ (or, in other notation, $Q' \in swap^{-1}(Q)$).*
- *Otherwise, $Q' = Q$ (up to isomorphism).*

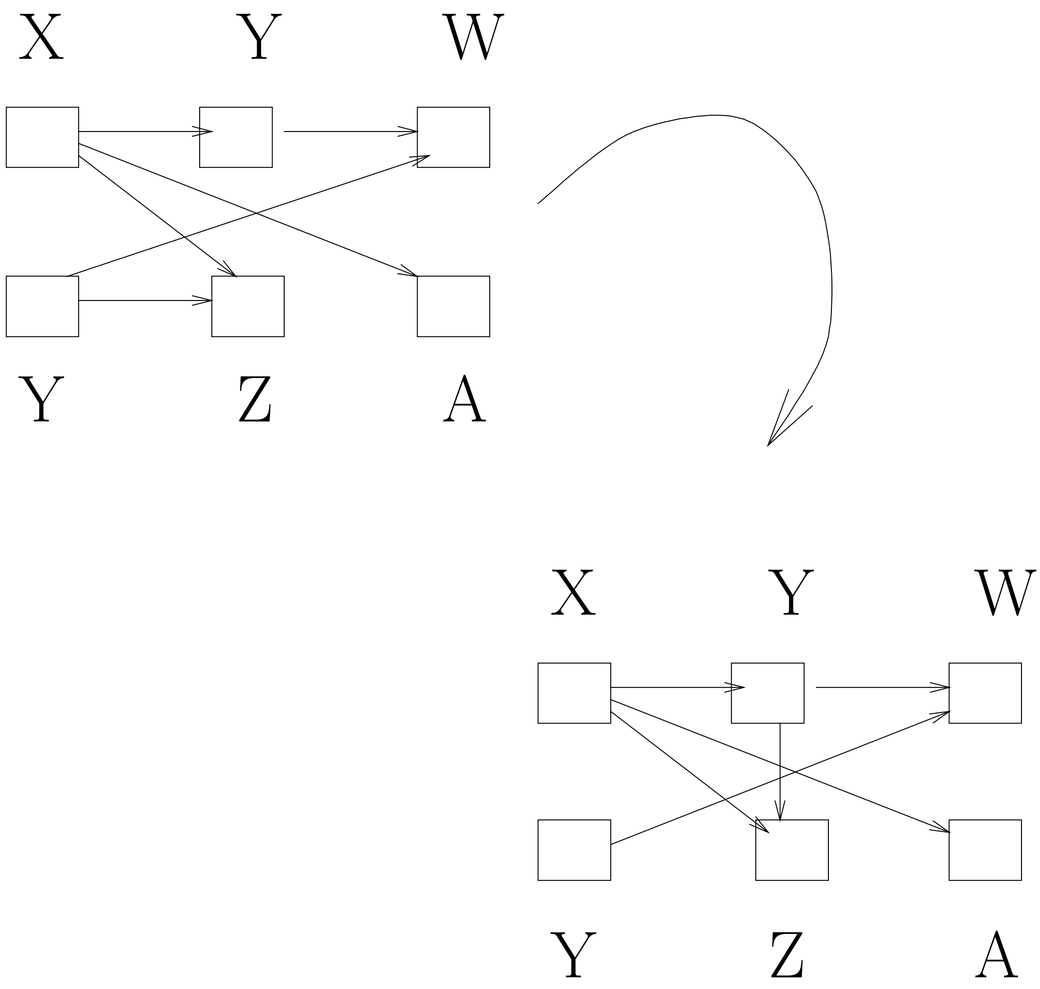


Figure 14: The swap operation: the pomset in the right is the result of swapping the two Y-labeled events in the left pomset.

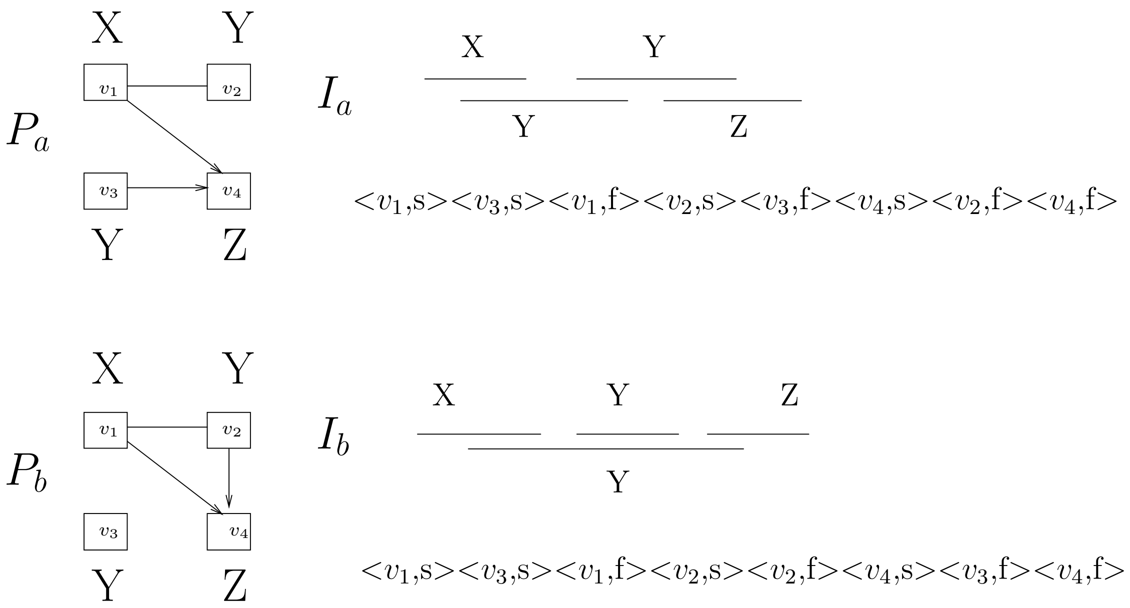


Figure 15: The string-equivalence of swapped pomsets:

Proof: Straightforward $\square$

Lemma 7.6 is an easy direction of Vogler's main result (Theorems 5.8 and 6.8). The other direction is much more difficult:

Notation : for a pomset P, we use $(S^*A)(P)$ for $swap^*(Aug(P) \cap Int)$ (where Int is the set of all interval pomsets). We use $(S^*A)^*(P)$ for the iterative closure of $(S^*A)(P)$, and define $(S^*A)^+ = (S^*A)^*(S^*A)$. For a pomset language PL, we define $(S^*A)^+(PL)$ as the union of $(S^*A)^+(P)$ for all $P \in PL$.

Definition 7.8 *(swap equivalence, Vogler [13]) Two pomset languages PL_1 and PL_2 are* swap-equivalent *if $(S^*A)^+(PL_1) = (S^*A)^+(PL_2)$.*

Theorem 7.9 *([13], theorem 6.8) Let P and Q be interval pomsets. If $\forall m.Q \bullet split_m \subseteq P \bullet split_m$, then $Q \in (S^*A)^+(P)$.*

A consequence of lemmas 7.6 and 7.9 is that split-equivalence coincides with the newly-defined swap-equivalence:

Theorem 7.10 *[13] Two pomset languages PL_1 and PL_2 are split-equivalent iff they are swap-equivalent.*

In the rest of this section we offer another proof to Theorem 7.9, based on our results. One could strengthen Theorem 3.4 and Lemma 3.8:

Theorem 7.11 *(strengthened Theorem 3.4) Given an interval pomset P then there exists a word s over $\Sigma \times Nat$ such that :*

1. $s \in P \bullet split_{max\_width(P)+1}$
2. *For any interval pomset Q if $s \in Q \bullet split_{max\_width(P)+1}$ then $P \in Aug(swap^*(Q))$.*

Lemma 7.12 *(strengthened Lemma 3.8) Given an interval pomset Q, a distinctive word s over $\Sigma \times Nat$ and a parsing h of s with respect to Q, and $change\_number(h) > 0$, then there exists a function h' and a pomset Q' such that:*

1. *h' is a parsing of s with respect to Q'*
2. $Q' \in swap^*(Q)$
3. $change\_number(h') < change\_number(h)$

The proof of Lemma 7.12 can be easily extracted from the proof of Lemma 3.8 (that appears in the next section). The derivation of Theorem 7.11 from Lemma 7.12 is the same as the derivation of Theorem 3.4 from Lemma 3.8.

Let us prove now Theorem 7.9. Let $wid = max\_width(P) + 1$. Assume that $\forall m. Q \bullet split_m \subseteq P \bullet split_m$. By Theorem 7.11 there exists a word $s \in P \bullet split_{wid}$ with the properties specified in the theorem. By hypothesis, we have $s \in Q \bullet split_{wid}$. By Theorem 7.11, $P \in Aug(swap^*(Q))$, and we proved Theorem 7.9.

8 Proof of Lemma 3.8

For the proof of Lemma 3.8, we would like to create h' from h by eliminating a change, (say between v_1 and v_2), thus satisfying condition 3 of the lemma. However, in order to satisfy condition 2 we would like that Q' will be isomorphic to Q or in $swap(Q)$, but not in $swap^{-1}(Q)$. For this v_1 and v_2 must not be in anti-swap configuration, so not any change will be proper for eliminating. The next definition describes the nature of the change that we would like to eliminate, while the following lemma gives the motivation for this definition.

Definition 8.1 *(strong change) Given a pomset P, a distinctive word s and a parsing h of s with respect to P, a change in h at place i is* **strong** *if $h^{-1}(h(i))$ starts before and finishes before $h^{-1}(h(i-1))$ (meaning that the first and last element of $h^{-1}(h(i))$ are before the first and last element of $h^{-1}(h(i-1))$, respectively).*

Lemma 8.2 *Let P be a pomset, s a distinctive word and h a characteristic parsing of s with respect to P. If there is a strong change in h at place i, then $h(i-1)$ and $h(i)$ are not in anti-swap-configuration in P.*

Proof: Assume that $h(i-1) = v_1$ and $h(i) = v_2$ are in anti-swap-configuration in P. Say that there exist v_0 and v_3 such that $v_0 <_P v_1$, $v_0 co_P v_2$, $v_1 <_P v_3$ and $v_2 co_P v_3$. Because h is characteristic, $h^{-1}(v_2)$ starts before and finishes after $h^{-1}(v_1)$, and this is a contradiction to the strong change in h. □

We finally turn to the proof of Lemma 3.8. W.l.g we can assume that h is characteristic: if not, we augment Q as needed until h is characteristic. Note that the augmentation does not violate condition 2.
Let us look at a strong change in h, say it occurs in place m (such exist by Lemma 9.1). Assume $h(m) = v_1$, $h(m-1) = v_2$.

Next we define h' by:

$$h'(i) = \begin{cases} v_1 & \text{if } i \geq m \text{ and } h(i) = v_2 \\ v_2 & \text{if } i \geq m \text{ and } h(i) = v_1 \\ h(i) & \text{otherwise} \end{cases}$$

We define Q' such that $v <_{Q'} v'$ iff $h'^{-1}(v)$ precedes $h'^{-1}(v')$. Clearly, h' is a characteristic parsing of s with respect to Q'. We prove part 2 of Lemma 3.8 by showing that $Q' \in Aug(swap(Q))$ or $Q' \in Aug(Q)$. For this we first show:

Lemma 8.3 *Q' is an augmentation of $simple\_swap(Q, v_1, v_2)$*

Proof: Assume that $v <_Q v'$. By hypothesis $h^{-1}(v)$ precedes $h^{-1}(v')$. We prove by the following cases:

- $v \notin \{v_1, v_2\}$, so by definition of h' also $h'^{-1}(v)$ precedes $h'^{-1}(v')$ and we have $v <_{Q'} v'$, which is sufficient.
- $v = v_1$. In this case, $v' \notin \{v_1, v_2\})$, because v_1 and v_2 are concurrent. Clearly, $h^{-1}(v_1)$ precedes $h^{-1}(v')$ (since $v = v_1$). Since the biggest index in $h^{-1}(v_1)$ is equal or larger than m, we have $h'^{-1}(v_2)$ precedes $h'^{-1}(v')$, and $v_2 <_{Q'} v'$, which is also sufficient.
- $v = v_2$. from similar arguments, $h'^{-1}(v_1)$ precedes $h'^{-1}(v')$, and $v_1 <_{Q'} v'$.

Now, with the last result, and in view of Lemma 7.7, it is enough to show that v_1 and v_2 are not in anti-swap configuration. However, by Lemma 8.2 this cannot be, since h is characteristic.

□

In order to prove part 3 of Lemma 3.8 we show that no new changes were added to the parsing h' (relative to h). Assume that h' contains a change at place i while h doesn't contain a change at this place. It is impossible that $i < m$ because then $h(i) = h'(i)$ and $h(i-1) = h'(i-1)$, and this implies a same change in h. It is also impossible that $i = m$ because h does contains a change at place m, So we assume that $i > m$.
We proceed by the following cases.

1. $h'(i-1) \notin \{v_1, v_2\}$ and $h'(i) \notin \{v_1, v_2\}$. In this case we have $h(i) = h'(i) \neq h'(i-1) = h(i-1)$ and we have a change in h at place i - contradiction.
2. $h'(i-1) = v_1$ and $h'(i) \notin \{v_1, v_2\}$. In this case $h(i) \neq h(i-1)$, because if $h(i) = h(i-1) = v_2$, then $h'(i) = v_1$ (note that $i - 1 >= m$) which is not the case. Again, we have a change in h at place i - contradiction.
3. $h'(i-1) = v_2$ and $h'(i) \notin \{v_1, v_2\}$: this case is symmetric to the previous one.
4. $h'(i-1) \notin \{v_1, v_2\}$ and $h'(i) = v_1$. Again, $h(i) \neq h(i-1)$, because if $h(i-1) = h(i) = v_2$, then $h'(i-1) = v_1$ which is not the case, so we have a change in h at place i.
5. $h'(i-1) = v_1$ and $h'(i) = v_2$. In this case $h(i-1) = v_2$ and $h(i) = v_1$, so $h(i) \neq h(i-1)$.We have a change in h at place i.
6. $h'(i-1) = v_2$ and $h'(i) = v_1$. This case is symmetric to the previous one.

This completes the proof of part 3 and of the whole Lemma 3.8.

9 The existence of a strong change in a parsing

In this section we prove the following lemma.

Lemma 9.1 *Given a pomset P, a distinctive word s over $\Sigma \times Nat$ and a parsing h of s with respect to P such that $change\_number(h) > 0$, then h contains a strong change.*

First note that a change can occur only in an increasing part. Therefore, Lemma 9.1 follows from the following lemma.

Lemma 9.2 *Suppose that we have a pomset P, a n-distinctive word s over $\Sigma \times Nat$ and a parsing h of s with respect to P such that $change\_number(h) > 0$. Let $s[i_a \dots i_b]$ be the first increasing part such that $h[i_a \dots i_b]$ contains more than one event. Note that this part is the first that contains a change. Let $i_{ch} \in [i_a, i_b]$ be the place of the last change in $s[i_a \dots i_b]$. Then the change at place i_{ch} is strong.*

To proceed, we need the following definitions.

Assume we have P, s and h as above. For an event v, we denote by $start(v)$ the smallest index i such that $h[i] = v$ (i.e. $min(h^{-1}(v))$). We denote by $finish(v)$ the largest index i such that $h[i] = v$ (i.e. $max(h^{-1}(v))$).

Let P be a pomset. Let s be a word, and let h be a parsing of s with respect to P. Given an index i, we define $progress_h(v, i)$ as the number of times that v appears in $h(1 \dots i)$.

Observe that if $finish(v) = i$, then i is the smallest index such that $progress_h(v, i) = n$. Similarly, if $start(v) = i$, then i is the smallest index such that $progress_h(v, i) = 1$

We make one more observation about the structure of an increasing part. Let $s[i_s \dots i_f]$ be an increasing part of a string s. If $i_s < i < j < i_f$ and $h[i] = h[j] = v$, then for all $k \in [i, j]$, $h[k] = v$. Thus, appearence of events in a parsing of an increasing part is "continues", and has the form $v_1 v_1 \dots v_1 v_2 v_2 \dots v_2 \dots v_m v_m \dots v_m$. Indeed, assume that there are two consecutive occurences of v_j at positions i_1 and i_2 and $i_2 > i_1 + 1$ is an increasing part. Then if the first occurence is labeled by A_l then the second is labeled by $A_{l+i_2-i_1}$. Therefore $h^{-1}(v_j)$ cannot be $A_1 \dots A_n$.

We now turn to the proof of lemma 9.2.

Assume that $h[i_{ch} - 1] = v_1$ and $h[i_{ch}] = v_2$. In order to show that the change at place i is strong (see definition of a strong change - Definition 8.1), we show the following:

1. $finish(v_2) < finish(v_1)$
2. $start(v_2) < start(v_1)$

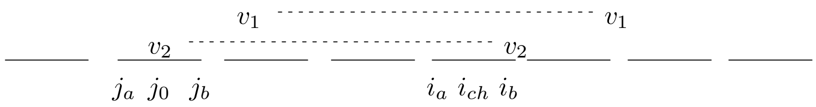


Figure 16: Existance of a strong change

Let us first show that $finish(v_2) < finish(v_1)$. We know that $s[i_{ch} - 1] = \langle A, k\rangle$ for $k = progress_h(v_1, i_{ch} - 1) < n$ (where A is the label of v_1 and v_2). Since $k < n$, it is clear that $finish(v_1) > i - 1$. Since the change at index i_{ch} is the last change in the part $s[i_a \dots i_b]$, we know that $h[i] = h[i+1] = \dots = h[i_b] = v_2$. Hence we have $progress_h(v_1, i_b) = progress_h(v_1, i_{ch}) < n$, so $finish(v_1) > i_b$. Because $s[i_b] = \langle A, n\rangle$, we have that $finish(v_2) = i_b$. To summerize, $finish(v_1) > i_b = finish(v_2)$.

Next we have to show that $start(v_2) < start(v_1)$. Observe that v_2 does not appear in $h[i_a \dots i{-}1]$. Hence $progress_h(v_2, i_a) = progress_h(v_2, i) > progress_h(v_1, i) > progress_h(v_1, i_a)$. Therefore, at place i_a we have that v_2 is "ahead" of v_1. We show that this is the case in all indices smaller than i_a. Formally, we show that for every decreasing or increasing part $s[j_a \dots j_b]$ of the string, such that $j_a \in [min(start(v_1), start(v_2)), i_a]$, we have $progress_h(v_2, j_a) > progress_h(v_1, j_a)$.
Assume that this is not the case. So there is a decreasing or increasing part $s[j_a \dots j_b]$ such that $progress_h(v_2, j_a) \leq progress_h(v_1, j_a)$ and $progress_h(v_2, j_b) > progress_h(v_1, j_b)$. Let j_0 be the largest index in $[j_a, j_b]$ for which $progress_h(v_2, j_0) \leq progress_h(v_1, j_0)$. Since *progress* values are increasing in steps of one (as i increases), we have that $progress_h(v_2, j_0) = progress_h(v_1, j_0) = l$. We proceed by two cases:

1. The part $s[j_a \dots j_b]$ is increasing. We show that this case cannot be since v_1 and v_2 do not appear in $h[j_a \dots j_b]$.
   By the choice of the part $s[i_a \dots i_b]$, we know that $h[j_a \dots j_b]$ consists of only one event v (see the assumption of Lemma 9.2). The event v cannot be neither v_1 nor v_2, since $finish(v) = j_b$. Thus, for all $j \in [j_a, j_b]$ we have $progress_h(v_2, j) = progress_h(v_1, j)$. This is a contradiction to the assumption that $progress_h(v_2, j_b) > progress_h(v_1, j_b)$.

2. The part $s[j_a \dots j_b]$ is a decreasing part. It is easy to see that for any label A, index i and $k \in Nat$, the value $capacity(A, s, i, k)$ is the number of events v labeled by A such that $progress_h(v, i) = k$. Hence, we have $capacity(A, s, j_0, l) \geq 2$. This is a contradiction to the definition of a decreasing part (see Definition 4.4).

We conclude that indeed $start(v_2) < start(v_1)$. Otherwise, we have that:
$1 = progress_h(v_1, start(v_1)) < progress_h(v_2, start(v_1)) = 0$.
This completes the proof of lemma 9.2.

10 The distinctive word is optimal

In this section we show that for every n there exist two pomset languages PL_1 and PL_2 such that:

1. $max(max\_width(PL_1), max\_width(PL_2)) = n$
2. $PL_1 \bullet split_m = PL_2 \bullet split_m$ for every $m \leq n$.
3. $PL_1 \bullet split_{n+1} \neq PL_2 \bullet split_{n+1}$.

Therefore showing that the choice of $max\_width(P) + 1$ in Theorem 3.1 is optimal. Examples for two pomset languages that satisfy conditions 1-3 also appear in [12]; We give our examples for the sake of completeness.
Our example uses the pomset $P^n = (V, <, lab)$, where:

$V = \{v_y^i | i = 1 \ldots n\} \cup \{v_x^i | i = 1 \ldots n-1\} \cup \{v_z^i | i = 2 \ldots n\}$

$\forall i = 1 \ldots n, j = 1 \ldots n-1, j \leq i, v_x^i < v_y^j$
$\forall i = 1 \ldots n, j = 2 \ldots n, j \leq i, v_y^i < v_z^j$
$\forall i = 1 \ldots n-1, j = 2 \ldots n, v_x^i < v_z^j$

$\forall i = 1 \ldots n, lab(v_x^i) = X$
$\forall i = 1 \ldots n, lab(v_y^i) = Y$
$\forall i = 1 \ldots n, lab(v_z^i) = Z$

Figure 17 shows the pomset P^n.

Consider the singleton pomset language $PL_1 = \{P^n\}$ on one hand, and the pomset language $PL_2 = swap^*(P^n) \cup Aug(P^n) \setminus \{P^n\}$ on the other. Note that $max\_width(PL_1) = n$ and $max\_width(PL_2) \leq n$

Lemma 10.1 *$PL_1 \bullet split_m \neq PL_2 \bullet split_m$, for some $m \leq n+1$*

Proof: First, we observe that $(S^*A)^+(PL_1) \neq (S^*A)^+(PL_2)$, since the (interval) pomset P^n itself does not belong to the right side (note that it *does* belong to the left side because it is an interval pomset). This is a consequence of the fact that a relation $\prec$ on pomsets, which is defined by: $P \prec Q$ iff $P \in (S^*A)^+(Q)$, is a partial order (see [13]).
By Theorem 7.10 we have that PL_1 and PL_2 are not split-equivalent. By the converse implication of Theorem 3.1, there exists some $m \leq n+1$ such that $PL_1 \bullet split_m \neq PL_2 \bullet split_m$. □

Lemma 10.2 $PL_1 \bullet split_n = PL_2 \bullet split_n$

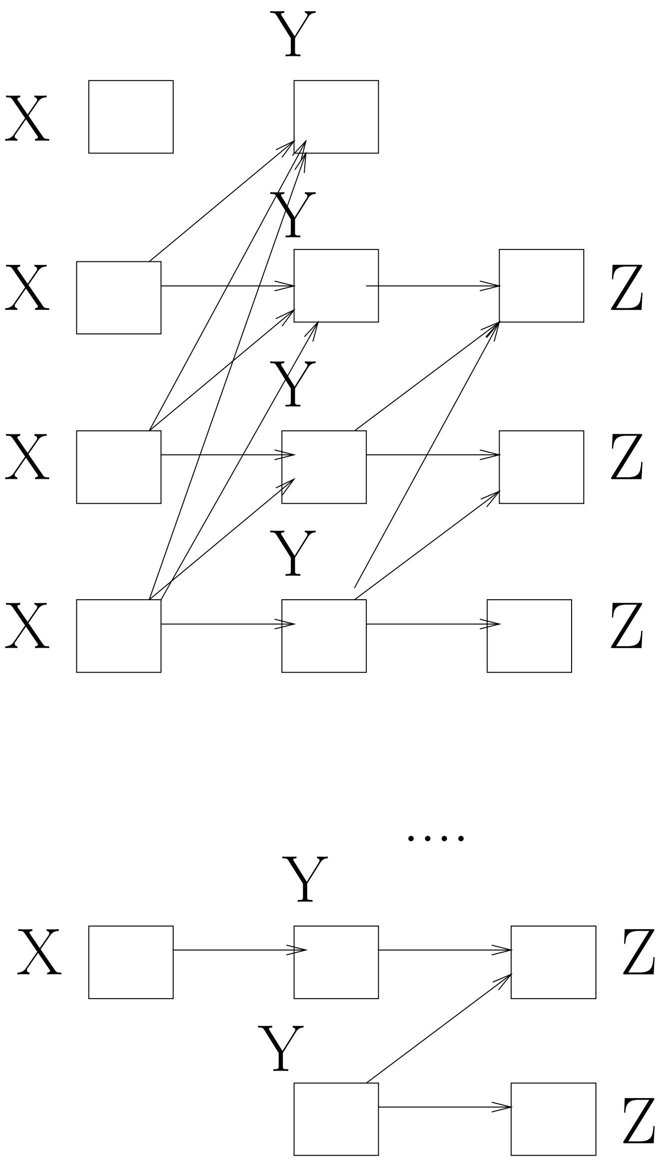
Proof: The $\supseteq$ direction: by definition, any $P' \in PL_2$, is in $Aug(P^n)$ or in $swap^*(P^n)$. If $P' \in Aug(P^n)$, then by Lemma 2.14, $\forall m.P' \bullet split_m \subseteq P^n \bullet split_m$. If $P' \in swap^*(P^n)$, then by Lemma 7.6, also $\forall m.P' \bullet split_m \subseteq P^n \bullet split_m$.
For the $\subseteq$ direction, assume that some word $w \in PL_1 \bullet split_n$. By hypothesis, $w \in P^n \bullet split_n$. We proceed by two cases:

- There exists a characteristic parsing h of w with respect to P^n. Then there is an index $0 < m < length(w)$ after which all events labeled with Y have started and none of them has finished. More formally, for every event v with label Y, $min(h^{-1}(v)) < m$ and $max(h^{-1}(v)) > m$. This fact implies, that for some index $0 < k < n$ we have $capacity(w, Y, k, m) \geq 2$. In other words, since n events have made a progress of 1 to $n-1$, there must be two events v_1 and v_2 for which $progress_h(v_1) = progress_h(v_2) = k$ such that $0 < k < n$.
  Similar to what is done in Lemma 3.8, we define a new parsing h' with respect to a new pomset P':
  $$h'(i) = \begin{cases} v_1 & \text{if } i \geq m \text{ and } h(i) = v_2 \\ v_2 & \text{if } i \geq m \text{ and } h(i) = v_1 \\ h(i) & \text{otherwise} \end{cases}$$
  We define P' such that $v <_{P'} v'$ iff $h'^{-1}(v)$ precedes $h'^{-1}(v')$.
  h' is clearly a parsing of w with respect to P', so $w \in P' \bullet split_n$. We would like to show that $P' \in PL_2$, thus showing that the word w is in $PL_2 \bullet split_n$. By Lemma 8.3, P' is an augmentation of $simple\_swap(P^n, v_1, v_2)$. Since v_1 and v_2 are in swap configuration in P^n (like any pair of events labeled by Y), we have that $P' \in (S^*A)^+(P^n)$, and by definition $P' \in PL_2$.

- If a characteristic parsing does not exist, we take some other parsing h and augment P^n until we get a pomset $P' \in Aug(P^n)$ such that h is a characteristic parsing of w with respect to P'. This is enough, since $P' \in PL_2$ by definition of PL_2.

$\square$

Note that the property of the pomset P^n that we used, is that every pair from the n events labeled by Y is in swap-configuration. Any other pomset with this property suits this example.



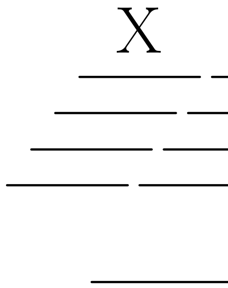


Figure 17: The pomset P^n. The relations between the left, X-labeled events to the right, Z-labeled ones have been omitted for the sake of readability. On the right, the set of intervals corresponding to P^n.

11 Conclusions and Related Topics

11.1 Expressions with constants

This paper considers constant-free expressions. However, the result is valid also for expressions with constants. We could consider an expression with constants, to which corresponds a pomset with two kinds of events: atomic events and refinable events. The split operation then splits only the refinable events, leaving the atomic events as they are.

A full proof of the validity of our result for expressions with constants is given in [1].

11.2 Expressions with intersection

In this paper we have considered only shuffle-regular expressions. When adding the intersection operator one can no longer work with pomsets, since intersection is not a **pomset definable operator** (see [6]). The decidability of language-equivalence between two intersection-shuffle-regular expressions was proven by Micciancio [5] using a very interesting semantics, which, unlike pomsets, captures also the intersection operation. Unfortunately, the proof does not allow expressions to contain constants, and the language-equivalence between expressions with intersection, shuffle, regular operations and constants remains open.

11.3 Using fixed alphabet

Another version of the problem is using fixed alphabet: given two expressions with variables and constants, are they equivalent (language or string equivalence) when using a fixed alphabet ?
For example, the expressions $(0+1)^*+X$ and $(0+1)^*$ are clearly not language-equivalence for any language, but are language-equivalence when restricting ourselves to the binary alphabet $\{0, 1\}$. The interesting thing is, that although this looks simple, the decidability of this equivalence is an open problem, even for regular expressions!
The proof in this paper, as well as the proofs in [6] and [5] uses an alphabet whose size is not bounded, but depends on the expressions. Therefore, these proofs does not resolve the fixed-alphabet version of the problems.

When we consider constant-free expressions, it can be shown (see [1]) that there is no difference between a fixed and an arbitrary alphabet. Thus, the fixed-alphabet equivalences are only interesting when applied to expressions with constants.

	Language equivalence			String equivalence		
	Regular	with Shuffle	with Shuffle & Intersection	Regular	with Shuffle	with Shuffle & Intersection
Without constants	"folk"	[3], [6]	[5]	"folk"	This Paper	Open Problem
With constants	"folk"	[6]	Open Problem	"folk"	This Paper	Open Problem
With constants, Fixed Alphabet	Open Problem	Open Problem	Open Problem	Open Problem	Open Problem	Open Problem

Figure 18: Decidability results.

11.4 Summary of results

All the results mentioned in the above subsections are about language equivalence. This paper is the first to determine decidability of string equivalence. As far as we know, string-equivalence decidability is still open for all the extensions mentioned in the above subsections: intersection and fixed alphabet.
Figure 18 summarizes the decidability results. The arrow in the figure expresses a non-trivial reduction that can be done (see [5]).

11.5 Vogler's lemma

Lemma 6.7 in [13] implicitly gives an algorithm to construct a "semi-distinctive" word from a pomset. That word is not "distinctive" in the sense of Definition 4.4, yet it satisfies the following condition: Let P be a pomset of size n and let s be a word constructed from lemma 6.7 in [13], then:

- $s \in P \bullet split_n$
- For every pomset Q, if $s \in Q \bullet split_n$ then $\forall m. P \bullet split_m \subseteq Q \bullet split_m$.

This condition is similar to the second condition in Theorem 3.4, but the size of the alphabet in this word is as the size of the pomset. From this lemma one can easily extract the decidability of split-equivalence between finite pomset languages, and therefore also the decidability of expressions with union, shuffle and concatenation (but no star). However it is not useful for pomset languages that contain infinite set of pomsets.

11.6 Generalization to (m,n)-equivalence

The results of this paper and the related ones can be generalized as follows: Two expressions E_1 and E_2 are (m,n)-equivalent (notation $E_1 \sim_{(m,n)} E_2$, where m and n possibly infinite) if they define the same language under instantiation of variables by languages of maximum m strings, each string of maximum length n. In this notation, *string-equivalence* coincides with $(1, \inf)$-equivalence, while *language-equivalence* coincides with $(\inf, \inf)$-equivalence (and, by theorem 1.1, also with $(\inf, n)$-equivalence for any $n \geq 2$). As for decidability of (m,n)-equivalence, we have the following:

- $\sim_{(\inf,2)} = \sim_{(\inf,3)} = \ldots = \sim_{(\inf,\inf)}$ was proved decidable in [6].
- $\sim_{(1,2)} \supsetneq \sim_{(1,3)} \supsetneq \ldots \supsetneq \sim_{(1,\inf)}$ was proved decidable in this paper.
- $\sim_{(m,n)}$ where $\inf > m > 1$ is finite, hence decidable.

Therefore, we are left with $\sim_{(m,\inf)}$, where $m > 1$. This equivalence decidability is still, as far as we know, an open problem. However, we have the following conjecture:
Given expressions E_1 and E_2, then they are $(m, \inf)$-equivalent iff they are $(m, max(sw(E_1), sw(E_2)) + 1)$-equivalent.
This conjecture is a generalization of Corollary 3.2. If it is correct, it implies decidability of $\sim_{(m,\inf)}$ for shuffle regular expressions.

11.7 Pomset equivalence

As mentioned in the introduction, Meyer and Rabinovich [6] proved the decidability of language equivalence for expressions. They have shown that the corresponding equivalence of pomsets is the **pomset-language-equivalence**: two pomset languages are pomset-language-equivalent if they are equivalent after refining each event by a pomset-language (this corresponds to instantiating every variable by a language).
String-equivalence, as shown in this paper, corresponds to split-equivalence between pomset-languages: two pomset-language are split-equivalent if they are equivalent after refining each event by a word, or a sequential linear pomset.
The first equivalence is after refining each event by a pomset-language. The second is after refining each event by a word. It seems, then, that we have another equivalence between the two: **pomset equivalence**, testing equivalence after refining each event by a pomset.
We will consider a slightly different version: we will allow replacement of the events not by any pomset, but only by **series-parallel (SP) pomsets**. The set of SP-pomsets is the minimal set of pomsets which contains the singleton pomset and is closed under concatenation and shuffle. The reader will observe that these pomsets correspond to expressions that are made of concatenation and shuffle only (=without union and star).

It was proved in [1] that series-parallel-pomset-equivalence (SPP-equivalence) coincides with split-equivalence.

Acknowledgments

The authors are grateful to Daniele Micciancio for his comments.

References

[1] Y.Abramson. Decidability of split-equivalence. M.Sc. Thesis, Tel-Aviv university, 1998.

[2] S.L.Bloom, Z.Esik. Free shuffle algebras in language varieties. Theoretical Computer Science 163 (1996) 55-98.

[3] J.L.Gischer. Partial Orders and Theory of Shuffle. PhD thesis, Stanford University, 1984.

[4] J.L.Gischer. The equational theory of pomsets. Theoretical Computer Science,61(2-3):199-224, November 1988.

[5] D.Micciancio. The Validity problem for Extended Regular Expressions. M. Sc. thesis, MIT, 1996.

[6] A. R. Meyer, A. Rabinovich A solution of an interleaving decison problem by a partial order techniques. In Proceedings of Workshop on Partial-Order Methods in Verification, July 1996, Princeton NJ, pp.203-212, Editors G. Holzmann, D. Peled, V. Pratt, AMS-DIMACS Series in Discrete Mathematics.

[7] M.Nielsen, U.Engberg and K.Larsen. A Simple Process Language with Refinement. In *REX Workshop on Linear Time, Branching time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lect. Notes in Computer Science*, pages 523-548, Springer Verlag, 1990.

[8] J.L.Peterson. Petri Net Theory. Prentice-Hall, 1981.

[9] V.R.Pratt, On the composition of Processes. In *Proc. of the Ninth Annual ACM Symposium on Principles of Programming Languages*, 1982.

[10] W.Reisig. Petri nets EATCS Monographs on Theoretical Computer Science. Springer, 1985.

[11] B.A.Trakhtenbrot and Y.M.Barzdin. Finite Automata, North Holland Amsterdam, 1973.

[12] R.van Glabbeek and F.Vaandrager, The difference between splitting in n and n + 1. Information and Computation, 136(2):109-142, 1 August 1997.

[13] W.Vogler. The limit of $Split_n$-language equivalence. Information and Computation,127(1):41-61, 25 May 1996.

[14] N.Wiener. A contribution to the theory of relative position. In *Proc. Camb. Philos. Soc.* 17, pp 441-449, 1914.