SECTION 8

RELATIONSHIP TO OTHER MODELS OF CONCURRENCY

A. Mazurkiewicz:  *Trace Theory*

G. Winskel:  *Event Structures*

M. Nielsen:  *CCS - and its Relationship to Net Theory*

E. Best:  *COSY: Its Relation to Nets and to CSP*

E. R. Olderog:  *TCSP: Theory of Communicating Sequential Processes*

W. Kluge:  *Reduction, Data Flow and Control Flow Models of Computation*
TRACE THEORY

Antoni Mazurkiewicz
Institute of Computer Science
Polish Academy of Sciences
P.O.Box 22, PL-00-901 Warsaw

ABSTRACT. The concept of traces has been introduced for describing non-sequential behaviour of concurrent systems via its sequential observations. Traces represent concurrent processes in the same way as strings represent sequential ones. The theory of traces can be used as a tool for reasoning about nets and it is hoped that applying this theory one can get a calculus of the concurrent processes analogous to that available for sequential systems. The following topics will be discussed: algebraic properties of traces, trace models of some concurrency phenomena, fixed-point calculus for finding the behaviour of nets, modularity, and some applications of the presented theory.

Key words: concurrency, traces, processes, partial ordering, Petri nets.

CONTENTS:

0. Introduction
1. Sequential languages
   1.1. Strings
   1.2. String languages
   1.3. String systems
2. Dependency and traces
   2.1. Concurrent alphabets
   2.2. Traces
   2.3. Dependency graphs
   2.4. Trace ordering
   2.5. Decomposition of traces
   2.6. Trace prefixes structure
2.7. Trace projections
3. Trace systems
  3.1. Trace languages
  3.2. Generalized dependency graphs
  3.3. Regular trace languages
  3.4. Fixed point equations
  3.5. Trace systems
  3.6. Synchronization
4. Nets and traces
  4.1. Elementary net systems
  4.2. Firing sequences
  4.3. Firing traces
  4.4. Composition and decomposition of systems
5. Traces and processes
  5.1. Trace state space
  5.2. Trace structures
  5.3. Observations
  5.4. Invariance and inevitability
6. Bibliographical notes
7. References

0. INTRODUCTION

In concurrent system activities the ordering of event occurrences is partial: there is no means to decide which one from independent events occurs earlier than the other. The only way to establish objective ordering of event occurrences in such systems is to find their mutual causal dependencies and to agree that a cause must be always earlier than its effect. Therefore, the dependency (or independency) of event occurrences should be a basis for a concurrent behaviour description. On the other hand, a single observer of a concurrent system notices a sequence of events occurring in the system; thus, it would be useful to combine sequential observations (expressed as strings of events) with a dependency relation (expressed as a relation in the set of event occurrences) in order to grasp some concurrency phenomena and to investigate the nonsequential behaviour of systems via their sequential observations.

The algebra of strings and of sets of strings (languages) has been proved to be of great importance for the theory of sequential systems. Many various models of
computations have been investigated by means of sets of strings they can generate (or accept). Such notions as finite automata, sequential machines, regular languages, regular expressions, are strongly connected with strings of symbols representing actions occurring in real systems, since the behaviour of such systems can be described by sets of interpreted strings. The algebra of strings has established a basis for all subsequent theoretical investigations.

In case of concurrent systems, strings of symbols, perfectly suited for modelling sequential processes, are not so generally and directly available. Since some events occurring in concurrent systems are independent of each other, there is no causal relationship putting them in one linearly ordered sequence; instead, their occurrences form a partially ordered set. Therefore an attempt should be made to adjust string oriented methods and techniques to the reality of concurrency.

The theory of traces was motivated by the above arguments. It started in 1977 as a tool for analysing the behaviour of Petri nets. Since then a considerable progress has been made in the theory itself as well as in its applications in related branches of concurrency theory, especially in the theory of Petri nets. On the other hand, trace theory is closely related to the algebraic theory of free partially commutative monoids which is recently intensively developed.

Nowadays, there are several research directions in developing the theory. One is connected with graph representation of traces and the theory of graph grammars; the aim is to overcome some limitations caused by strings and to deal with explicit partial ordering of event occurrences. The second, algebraic, is connected with theory of trace languages, as a basis for concurrent systems specification and description techniques. Issues connected with trace models for temporal logic lead to another research direction.

The paper aims to present briefly some facts and methods offered by trace theory. The proofs of theorems are not included; they are either obvious, or can be found elsewhere. The paper does not pretend to give the whole review of the theory; it would be impossible within the limited size of the paper. The choice of topics reflects the personal interest of the author. The enclosed bibliography gives references to source papers connected with trace theory and those references to papers on Petri nets which are directly related to the topics discussed here.

The standard mathematical denotations are used through the paper. The set of nonnegative integers is denoted by \( \mathbb{N} \). The braces around singletons are omitted.
1. SEQUENTIAL LANGUAGES

1.1. Strings. Let $A$ be a finite set of symbols (an alphabet). The ordered triple $S = (A^*, \cdot, \varepsilon)$, where $A^*$ is the set of all finite sequences (strings) of symbols in $A$, $\varepsilon$ is the empty string, and $\cdot$ is the concatenation operation on strings, is the well known monoid of strings over $A$ (usually the sign $\cdot$ for concatenation is omitted in expressions). Interpreting symbols in alphabet $A$ as elementary (atomic) actions, strings over $A$ can be viewed as composed actions; the effect of such a composed action is the join effect of the constituent atomic actions executed in the sequence indicated by the string.

For each string $w \in A^*$ define

$$\text{Rng}(w) = \{a \in A \mid \exists w_1, w_2 \in A^*: w = w_1aw_2\};$$

$\text{Rng}(w)$ is called the range of $w$. Clearly, for each string $w$ over an alphabet $A$ we have $\text{Rng}(w) \subseteq A$ but not necessarily $\text{Rng}(w) = A$.

With each string an ordered set can be associated, namely that of symbol occurrences. Let $w = a_1a_2...a_n$ be a string, $n \geq 0$; each ordered pair $(a_i, n_i)$ with $n_i = \text{card}()$ 1 $j$, $a_i = a_j$ is called an occurrence of $a_i$ in $w$. The relation

$$\text{Ord}(w) = \{(a_i, n_i), (a_j, n_j) \mid 1 \leq i \leq j \leq n\}$$

is an ordering relation. This ordering will be referred to as the string ordering of $w$ and denoted by $\text{Ord}(w)$; it can be interpreted as the ordering of actions execution during a single run of a (sequential) system. Clearly, $\text{Ord}(w)$ is a linear ordering for each string $w$.

Let $w$ be a string over an alphabet $A$. Elements of the set

$$\text{Pref}(w) = \{u \mid \exists v \in A^*: uv = w\}$$

are called prefixes of $w$. Obviously, $w \in \text{Pref}(w)$ and $\varepsilon \in \text{Pref}(w)$ for each string $w$. Let $\leq$ be a relation in $A^*$ such that
Clearly, $\leq$ is an ordering relation in $A^*$. This ordering will be called the **prefix ordering** of strings. For each string $w$ the set $\text{Pref}(w)$ is linearly ordered by the prefix ordering. If a string $w$ is interpreted as a composed action execution, elements of $\text{Pref}(w)$ can be viewed as all partial executions of that (composed) action; since each such partial execution uniquely determines a state of the system, elements of $\text{Pref}(w)$ can be viewed as well as (intermediate) states of the system arising during action $w$ execution.

Let $A$ be an alphabet, $w \in A^*$. For any alphabet $B$ denote by $w/B$ the (string) **projection** of $w$ onto $B$, i.e. a string over $A \cap B$ defined as follows:

- $\epsilon/B = \epsilon$,
- $(wa)/B = (w/B)a$, if $w \in A^*$, $a \in B$,
- $(wa)/B = (w/B)$, if $w \in A^*$, $a \notin B$.

Intuitively, projection from $A$ onto $B$ "erases" from a string over $A$ all symbols not belonging to $B$.

Let $u$, $v$ be strings; any string

$$u_1v_1u_2v_2\ldots u_nv_n, \ n \geq 0,$$

such that $u_1u_2\ldots u_n = u$, $v_1v_2\ldots v_n = v$, is called a **shuffle** of $u$ with $v$. The set of all shuffles of $u$ with $v$ is denoted by $\text{Shuffle}(u,v)$.

### 1.2. String languages

Let $A$ be an alphabet; any subset of $A^*$ is called a (string) **language** over $A$. If $L_1$, $L_2$ are languages over $A$, then their **concatenation** $L_1L_2$ is defined as

$$L_1L_2 = \{uv \mid u \in L_1, v \in L_2\}.$$

If $w$ is a string, $L$ is a language, then (according to our convention of omitting braces around singletons)

$$wL = \{wu \mid u \in L\}, \quad Lw = \{uw \mid u \in L\}.$$

The **power** of a language $L$ is defined recursively:
\[ L^0 = \{\varepsilon\}, \]
\[ L^{n+1} = L^n L, \]

for each \( n \in \mathbb{N} \). The iteration \( L^* \) of \( L \) is the union:

\[ \bigcup_{n \in \mathbb{N}} L^n. \]

For any language \( L \) over \( A \)

\[ \text{Pref}(L) = \bigcup_{w \in L} \text{Pref}(w). \]

Elements of \( \text{Pref}(L) \) are called prefixes of \( L \). Obviously, \( L \subseteq \text{Pref}(L) \). A language \( L \) is prefix closed, if \( L = \text{Pref}(L) \). Clearly, for any language \( L \) the language \( \text{Pref}(L) \) is prefix closed. An ordered set \((X, \preceq)\) is a tree, if:

(i) for each \( x \in X \) the set \( \{y \in X \mid y \preceq x\} \) is linearly ordered;
(ii) for each \( x, y \in X \) there is \( z \in X \) with \( z \preceq x, z \preceq y \).

PROPOSITION 1. For each prefix closed string language \( L \) the ordered set \((L, \preceq)\) is a tree. \( \Box \)

A language \( L \) is directed, if for each \( u, v \in L \) there is \( w \in L \) such that \( u \preceq w, v \preceq w \).

PROPOSITION 2. Any directed string language is linearly ordered by the prefix relation. \( \Box \)

EXAMPLE 1. The prefix ordering of languages \( \text{Pref}((ab)^*) \) and \( \text{Pref}((ac)^*ad) \) are presented in Fig. 1 as diagrams (1) and (2); both languages are prefix closed; the first is directed; it represents the infinite sequence \((a, b, a, b, a, b, \ldots)\). Both languages will be discussed later on. \( \Box \)

Extend the projection function from strings to languages defining for each language \( L \) over an alphabet \( A \) and any alphabet \( B \) the projection of \( L \) onto \( B \) as a language \( L/B \) over \( A \cap B \) defined as

\[ L/B = \{u/B \mid u \in L\}. \]

Extend also the notion of shuffling from strings to languages defining for arbitrary
1.3. String systems. Call a string system, or system for short, any ordered pair \((A, L)\) where \(A\) is an alphabet, \(L \subseteq A^*\), and \(L = \emptyset\) implies \(A = \emptyset\). If \(X = \{A, L\}\) is a string system, \(A\) is the alphabet of \(X\) denoted by \(\alpha(X)\) and \(L\) is the language of \(X\) denoted by \(\lambda(X)\). The difference between string languages and string systems is that different systems can have identical languages, as e.g. systems \((\{a\}, a^*)\), \((\{a, b\}, a^*)\). While a language gives information which symbols are present in strings of the language, a string system gives also information which symbols are absent in those strings; in other words, within systems represented by string systems it is possible to speak not only about executions of some actions but also about their omissions.

Let \(X\) be a string system. The prefix ordering of \(\lambda(X)\) will be called the prefix
ordering of $X$ and the diagram of this ordering will be called the diagram of $X$.

Denote by $\emptyset$ and call empty the system $(\emptyset, \emptyset)$. For each system $X$ and each string $w$ over $\alpha(X)$ define

$$wX = (\alpha(X), w\lambda(X)),$$

$$Xw = (\alpha(X), \lambda(X)w);$$

and extend this definition for string languages over $\alpha(X)$:

$$LX = (\alpha(X), L\lambda(X)),$$

$$XL = (\alpha(X), \lambda(X)L)$$

for each string language $L$ over $\alpha(X)$ and each string system $X$.

We say that a system $X$ is contained in another system $Y$ and write $X \subseteq Y$, if $\alpha(X) \subseteq \alpha(Y)$ and $\lambda(X) \subseteq \lambda(Y)$. Denote the class of all string systems by $\tilde{\alpha}$. Define merging as a binary operation $\&$ on string systems:

$$\&: \tilde{\alpha} \times \tilde{\alpha} \longrightarrow \tilde{\alpha}$$

such that for all string systems $X$, $Y$

$$\alpha(X \& Y) = \alpha(X) \cup \alpha(Y),$$

$$\lambda(X \& Y) = (w \in (\alpha(X \& Y))^* | \ w/\alpha(X) \in \lambda(X), \ w/\alpha(Y) \in \lambda(Y))$$

(we use here the infix notation and assume $\&$ to bind weaker than the concatenation and stronger than other set-theoretical operations). Let $X$, $Y$, $Z$ be string systems. The following equalities hold:

$$X \& X = X,$$

$$X \& Y = Y \& X,$$

$$X \& (Y \& Z) = (X \& Y) \& Z$$

$$((\epsilon, A) \& (\epsilon, B) = (\epsilon, A \cup B),$$

$$X \& \emptyset = \emptyset,$$

$$(X \cup Y) \& Z = X \& Z \cup Y \& Z.$$

Observe that for any string systems $X$, $Y$

$$\alpha(X) = \alpha(Y) \implies \lambda(X \& Y) = \lambda(X) \cap \lambda(Y),$$
\[ x(X) \cap x(Y) = \emptyset \implies \lambda(X \& Y) = \text{Shuffle}(\lambda(X), \lambda(Y)). \]

Note also that if \( \lambda(X) \), \( \lambda(Y) \) are singletons, then in general it is not so in case of \( \lambda(X \& Y) \). \( \lambda(X \& Y) \) is not empty if and only if

\[ \lambda(Y)/\mu(X) = \lambda(X)/\mu(Y). \]

**EXAMPLE 2.** Let \( X, Y \) be string systems,

\[
X = \{ (a,b),(ab)^*(a \cup \varepsilon) \},
\]

\[
Y = \{ (a,c,d),(ac)^*(ad \cup a \cup \varepsilon) \};
\]

observe that \( \lambda(X) = \text{Pref}((ab)^*) \), \( \lambda(Y) = \text{Pref}((ac)^*ad) \) (cf. Example 1). The merging of \( X \) and \( Y \) is the system

\[
X \& Y = \{ (a,b,c,d),(abc \cup acb)^*(abd \cup aab \cup ab \cup ac \cup ad \cup a \cup \varepsilon) \}.
\]

The diagram of \( X \& Y \) is presented in Fig.2 (cf. diagrams from Example 1). \( \Box \)

2. DEPENDENCY AND TRACES

2.1. Concurrent alphabets. By a **concurrent alphabet** we shall mean any ordered pair \( \Sigma = (A,D) \) where \( A \) is a finite set (the alphabet of \( \Sigma \)) and \( D \) is a symmetric and reflexive binary relation in \( A \) (the dependency in \( \Sigma \)). In particular, \( (\emptyset,\emptyset) \) (the empty alphabet), \( (A,\text{id}_A) \) and \( (A,A^2) \) (alphabets with minimum and full dependency) are concurrent alphabets for each alphabet \( A \). For each concurrent alphabets \( \Sigma_1 = (A_1,D_1) \), \( \Sigma_2 = (A_2,D_2) \), define

\[
\Sigma_1 \cup \Sigma_2 = (A_1 \cup A_2, D_1 \cup D_2),
\]

\[
\Sigma_1 \cap \Sigma_2 = (A_1 \cap A_2, D_1 \cap D_2),
\]

\[
\Sigma_1 \subseteq \Sigma_2 \iff A_1 \subseteq A_2 \text{ and } D_1 \subseteq D_2.
\]

It is clear that \( \Sigma_1 \cup \Sigma_2 \) and \( \Sigma_1 \cap \Sigma_2 \) are concurrent alphabets, and

\[
\Sigma_1 \cap \Sigma_2 \subseteq \Sigma_1 \subseteq \Sigma_1 \cup \Sigma_2, \quad i = 1,2.
\]
For any alphabet $A$ by $D(A)$ we shall denote the concurrent alphabet $(A, A^2)$ and call it the full dependency alphabet (over $A$). If $A = \{a_1, a_2, \ldots, a_n\}$, we shall write

$$D(a_1, a_2, \ldots, a_n)$$

instead of $D(A)$. It is clear that each concurrent alphabet is the union of a finite number of full dependency alphabets. Define $I_\Sigma = A^2 \setminus D$; $I_\Sigma$ will be called the independency in $\Sigma$; obviously, $I_\Sigma$ is a symmetric and irreflexive relation in $A$. Two symbols $a, b$ in $A$ will be called mutually dependent in $E$, if $(a, b) \in D$; otherwise they are said to be independent in $E$. Any triple $(A, I, D)$ where $A$ is an alphabet, $I \subseteq A^2$ is symmetric and irreflexive, and $D = A^2 \setminus I$ is called a reliance alphabet; this construct is sometimes used as a primitive concept of trace theory and then the concurrent alphabet $(A, D)$ is a derived notion.

Example 3. The concurrent alphabet $((a, b, c), (a, b)^2 \lor (a, c)^2)$ is the union of $D(a, b)$ and $D(a, c)$; symbols $b, c$ are (the only) independent symbols in this alphabet. $\square$
2.2. Traces. Let $\Sigma$ be a concurrent alphabet. Define $\Xi^{\Sigma}$ as the least congruence in the monoid $(A^*,\cdot,\varepsilon)$ such that

$$(a,b) \in I^{\Sigma}_\Xi \Rightarrow ab \equiv^{\Sigma} ba.$$

In other words, $w' \equiv^{\Sigma} w''$ if and only if there is a finite sequence of strings

$$(w_0,w_1,\ldots,w_n),$$

$n \geq 0$, such that

$$w_0 = w', \ w_n = w'',$$

and for each $i$, $1 \leq i \leq n$, there are strings $u$, $v$, and symbols $a$, $b$, $(a,b) \in I^{\Sigma}_\Xi$, with

$$w_{i-1} = uabv, \ w_i = ubav.$$

The relation $\equiv^{\Sigma}$ will be referred to as the trace equivalence over $\Sigma$.

Equivalence classes of $\Xi^{\Sigma}$ relation are called traces over $\Sigma$; a trace generated by a string $w$ will be denoted by $[w]^{\Sigma}$ (or by $[w]$, if $\Sigma$ is understood), and for each $L \subseteq A^*$ define

$$[L]^{\Sigma} = \{[w]^{\Sigma} \mid w \in L\}.$$

The set

$$\Theta(\Sigma) = [A^*]^{\Sigma}$$

is the set of all traces over $\Sigma$; if $\Sigma$ is known, we shall write $\Theta$ instead of $\Theta(\Sigma)$.

By definition, a single trace arises by identifying all strings which differ only in the ordering of adjacent independent symbols. If symbols in $A$ are thought as atomic system actions, traces are representing composed actions in which some elementary actions occur independently (i.e., without any causal relationship) of each other. The notion of traces allows to eliminate from strings the ordering between some process actions occurring independently of each other.

**EXAMPLE 4.** Let $\Sigma = D(a,b) \cup D(a,c)$. Then $b$, $c$ are independent in $\Sigma$; the equivalence
class generated by string abbaa is:

\[ \{\text{abbaa, abcba, acbba}\} \]

The quotient algebra \( T = (A^*, *, \epsilon)/\sim_\Sigma \) is called the algebra of traces over \( \Sigma \). Clearly, \( T \) is a monoid in which the concatenation operation \( * \) may be commutative for some different symbols (in contrast to the monoid of strings, where \( a * b \) implies \( ab \neq ba \)). The monoid \( T \) is sometimes called a free partially commutative monoid over \( A \). It is clear that in case of full dependency (\( D = A^2 \)) \( T \) is isomorphic to the ordinary algebra \( S = (A^*, *, \epsilon) \) of strings over \( A \). Now, we are going to develop the algebra of traces along the same lines as it was presented above in case of the algebra of strings.

Let \( \Sigma \) be a fixed concurrent alphabet. It is clear that \( [w_1] = [w_2] \) implies \( \text{Rng}(w_1) = \text{Rng}(w_2) \); thus, for all strings \( w \) we can define

\[ \text{Rng}([w]) = \text{Rng}(w). \]

The following equalities are direct consequences of the definition of traces:

\[ [u][w] = [uw], \]
\[ \tau(\epsilon) = [\epsilon]\tau = \tau, \]
\[ \tau_1(\tau_2\tau_3) = (\tau_1\tau_2)\tau_3, \]
\[ \sigma'\tau_1\sigma'' = \sigma'\tau_2\sigma'' \Leftrightarrow \tau_1 = \tau_2; \]

where \( u, w \) are strings, \( \tau, \tau_1, \tau_2, \tau_3, \sigma', \sigma'' \) are traces. The following equivalence for traces is a generalization of the well-known Levi lemma for strings:

\[ \alpha = \gamma \Leftrightarrow \exists \tau_1, \tau_2, \tau_3, \tau_4: \]
\[ \alpha = \tau_1\tau_2, \beta = \tau_3\tau_4, \gamma = \tau_1\tau_3, \delta = \tau_2\tau_4, \]
\[ \text{Rng}(\tau_2) \times \text{Rng}(\tau_3) \subseteq 1_\Sigma, \]

for any traces \( \alpha, \beta, \gamma, \delta \). In particular, if \( a, b \in A \), we have

\[ \tau_1[a] = \tau_2[b], a \neq b \Leftrightarrow [ab] = [ba] \text{ and } \exists \sigma: \tau_1 = \sigma[b], \tau_2 = \sigma[a] \]

for each traces \( \tau_1, \tau_2 \).
2.3. Dependency graphs. Dependency graphs are thought as graphical representations of traces, which make explicit the ordering of symbol occurrences within traces. It turns out that, for a given concurrent alphabet, algebra of traces is isomorphic with the algebra of dependency graphs defined below. Therefore, it is only a matter of taste which objects are chosen for representing concurrent processes: equivalence classes of strings or labelled graphs.

Let $\Sigma = (A, D)$ be a concurrent alphabet. Dependency graphs over $\Sigma$ (or d-graphs for short) are finite, oriented, acyclic graphs with nodes labelled with symbols from $A$ in such a way that two nodes of a graph are connected with an arc if and only if they are different and labelled with dependent symbols. Formally, a triple

$$G = (V, R, \varphi)$$

is a dependency graph, (d-graph) over $\Sigma$, if

- $V$ is a finite set (of nodes of $G$),
- $R \subseteq V \times V$ (the set of arcs of $G$),
- $\varphi : V \rightarrow A$ (the labelling of $G$),

such that

$$R^+ \cap \text{id}_V = \emptyset,$$
$$R \cup R^{-1} \cup \text{id}_V = \varphi D \varphi^{-1}.$$

Two d-graphs are isomorphic, if there exists a bijection between their nodes preserving labelling and arc connection. As usual, two isomorphic graphs are identified; all properties of d-graphs are given up to isomorphism.

EXAMPLE 5. Let $\Sigma = D(a, b) \cup D(a, c)$. Then the node labelled graph $(V, R, \varphi)$, with

$$V = \{1, 2, 3, 4, 5\}$$
$$R = \{(1, 2), (1, 3), (1, 4), (1, 5), (2, 4), (2, 5), (3, 5), (4, 5)\},$$
$$\varphi(1) = a, \varphi(2) = b, \varphi(3) = c, \varphi(4) = b, \varphi(5) = a,$$

is a d-graph. It is (isomorphic to) the graph in Fig. 3. $\square$

Let $w \in A^*$, $\Sigma$ be a concurrent alphabet; by $\langle w \rangle_\Sigma$ (or by $\langle w \rangle$, if $\Sigma$ is understood) denote the node labelled oriented graph defined recursively:
\( \langle \varepsilon \rangle \) is the empty graph (no nodes, no arcs);
\( \langle w \rangle e \) is a graph arising from \( \langle w \rangle \) by adding to it a new node labelled with \( e \) and new arcs, leading to this new node from all nodes of \( \langle w \rangle \) labelled with symbols dependent on \( e \), for each string \( w \) and symbol \( e \).

It is clear that the graph \( \langle w \rangle \) defined as above is a d-graph for any string \( w \) in \( A^* \). Conversely, it turns out that for any d-graph \( G \) over \( \Sigma \) there is a string \( w \in A^* \) such that \( G \) is isomorphic with \( \langle w \rangle \).

Therefore,

\[
\Gamma_\Sigma = \{ \langle w \rangle_\Sigma \mid w \in A^* \}
\]

is the class of all d-graphs over \( \Sigma \).

---

**Fig. 3:** Dependency graph of \([abbaa]\) with \( bc = cb \)

The relationship between traces and d-graphs over \( \Sigma \) is expressed by the following theorem.

**Theorem 1.** For each strings \( w', w'' \) in \( A^* \)

\[
[w']_\Sigma = [w'']_\Sigma \text{ iff } \langle w' \rangle_\Sigma \cong \langle w'' \rangle_\Sigma,
\]

where \( \cong \) denotes the label preserving isomorphism of d-graphs. \( \Box \)
2.4. Trace ordering. Let $g = (V, R, \phi)$ be a d-graph over $\Sigma$. Let

$$\xi: V \rightarrow A \times \omega$$

be a function defined as follows:

$$\xi(v) = (\phi(v), \#(v)),$$

where

$$\#(v) = \text{card} \{u \mid \phi(u) = \phi(v), (u, v) \in R\} + 1.$$

Thus, $\xi$ assigns to each node of $g$ an occurrence of its label, i.e. an occurrence of a symbol from $A$. Since each d-graph determines a partial ordering of its nodes, and $\xi$ is a one-to-one function, $g$ determines (via $\xi$) a partial ordering of symbol occurrences. Denote this ordering by $\text{Ord}(g)$; for each string $w$ call $\text{Ord}(\langle w \rangle)$ the **trace ordering** in $[w]$.

**Theorem 2.** For each string $w$: $\text{Ord}(\langle w \rangle) = \bigcap_{u \in [w]} \text{Ord}(u)$. $\Box$

It means that the string orderings in representants of a trace $\tau$ are linearizations (extensions to the total ordering) of the trace ordering within $\tau$. In other words, the trace ordering in a trace is the greatest ordering common for all its representants.

Define in $\Gamma_\Sigma$ the composition operation $*$ as follows: for each graphs $g_1$, $g_2$ in $\Gamma_\Sigma$ the composition $(g_1 * g_2)$ of $g_1$ with $g_2$ is a graph arising from the disjoint union of $g_1$ and $g_2$ by adding to it new arcs leading from each node of $g_1$ to each node of $g_2$ provided they are labelled with dependent symbols.

**Example 6.** Let $\Sigma = D(a, b) \cup D(a, c)$. In Fig. 4 the composition of dependence graphs $\langle abb \rangle, \langle ca \rangle$ is depicted; the dotted arrows represent arcs added in the composition. $\Box$

**Theorem 3.** Given a concurrent alphabet $\Sigma = (A, D)$; the algebra $(\Gamma_\Sigma, *, \langle \epsilon \rangle_\Sigma)$ of dependency graphs over $\Sigma$ is isomorphic with the algebra $(\text{Tr}(\Sigma), *, [\epsilon]_\Sigma)$ of traces over $\Sigma$; the isomorphism is established by the correspondence

$$\langle w \rangle_\Sigma \leftrightarrow [w]_\Sigma$$
for each string \( w \in A^* \). \( \square \)

The dependency graph corresponding to a trace \( \tau \) will be denoted by \( \delta(\tau) \).

\[ \begin{array}{c}
\text{Fig. 4: Composition of dependency graphs}
\end{array} \]

2.5. Decomposition of traces. Call two traces \( \tau_1, \tau_2 \) over \( \Sigma \) independent, if

\[ \text{Rng}(\tau_1) \times \text{Rng}(\tau_2) \subseteq I_\Sigma, \]

and dependent otherwise. A trace \( \tau \) over \( \Sigma \) is connected, if any nonempty traces \( \tau_1, \tau_2 \) with \( \tau = \tau_1 \tau_2 \) are dependent. In other words, \( \tau \) is connected if \( \delta(\tau) \) is a connected graph.

Assign to each trace \( \tau \) over \( \Sigma \) a set \( \Delta_\Sigma(\tau) = \{\tau_1, \tau_2, \ldots, \tau_n\} \) of traces over \( \Sigma \) such that

\[ \begin{align*}
\tau &= \tau_1 \tau_2 \ldots \tau_n, \\
\tau_1 &\text{ is nonempty and connected for each } i, 1 \leq i \leq n, \\
\tau_i, \tau_j &\text{ are independent for each } i, j, 1 \leq i < j \leq n.
\end{align*} \]

Elements of \( \Delta_\Sigma(\tau) \) will be called independent components of \( \tau \), and the set \( \Delta_\Sigma(\tau) \) the decomposition of \( \tau \). If \( \Sigma \) is known, we shall write \( \Delta(\tau) \) rather than \( \Delta_\Sigma(\tau) \).

EXAMPLE 7. Let \( \Sigma = D(a) \cup D(b) \); then

\[ \Delta([abbab]) = \{[aa], [bbb]\}. \quad \square \]
It is clear that in case of full dependency $\Delta(\tau) = \{\tau\}$ for each trace $\tau$.

2.6. Trace prefixes structure. Let $\Sigma$ be a concurrent alphabet. A trace $\sigma$ is a prefix of a trace $\tau$ over $\Sigma$, if there is $\gamma$ such that $\tau = \sigma\gamma$. The set of all prefixes of a trace $\tau$ will be denoted by $\text{Pref}(\tau)$. The set of all traces over $\Sigma$ is ordered by a relation $\subseteq \Theta(\Sigma) \times \Theta(\Sigma)$ defined as

$$\tau_1 \subseteq \tau_2 \iff \tau_1 \in \text{Pref}(\tau_2).$$

Relation $\subseteq$ will be referred to as the dominating relation in $\Theta(\Sigma)$ and if $\tau_1 \subseteq \tau_2$ we say that $\tau_1$ is dominated by $\tau_2$ or that $\tau_2$ dominates $\tau_1$. In contrast to linearly ordered prefixes of a single string the set $\text{Pref}(\tau)$ is ordered by $\subseteq$ only partially.

![Dependency graph and prefix structure](image)

**Fig. 5:** Dependency graph (a) and prefix structure (b) of [abbca] with bc $\equiv$ cb

**EXAMPLE 8.** Let $\Sigma = D(a,b) \cup D(a,c)$. In Fig. 5 the dependency graph of trace [abbca] and the prefix structure of [abbca] are presented. In both diagrams the arrows following from others by transitivity or reflexivity are omitted. □
While the ordering given by the dependency graph of a trace can be interpreted as the ordering of action occurrences in a process, that given by the prefix ordering of the same trace can be viewed as the (global) state ordering in this process; \( \varepsilon \) represents the initial state; incomparable prefixes represent states arising in effect of the concurrent actions execution; if \( \tau_2 \) dominates \( \tau_1 \), \( \tau_1 \) is the effect of a partial execution of a process leading to \( \tau_2 \).

2.7. Trace projections. Let \( \Sigma_1, \Sigma_2 \) be concurrent alphabets, \( \Sigma = \Sigma_1 \cup \Sigma_2 \), \( A \) be the alphabet of \( \Sigma \), \( \tau \) be a trace over \( \Sigma_1 \), \( \tau = [w] \). Denote by \( \tau/\Sigma_2 \) the (trace) projection of \( \tau \) onto \( \Sigma_2 \), i.e. a trace over \( \Sigma \), defined by the following equality

\[
[w]_{\Sigma_1}/\Sigma_2 = [w/\Lambda]_{\Sigma}.
\]

It can be easily checked that the above definition is correct, i.e. that if \([w]_{\Sigma_1} = [u]_{\Sigma_1}\), then \([w]_{\Sigma_1}/\Sigma_2 = [u]_{\Sigma_1}/\Sigma_2\).

Let \( \Sigma_1, \Sigma_2 \) be concurrent alphabets, \( \Sigma = \Sigma_1 \cup \Sigma_2 \). The following equalities hold:

\[
\begin{align*}
\tau/(\emptyset, \emptyset) &= [\varepsilon]/(\emptyset, \emptyset), \\
(\tau / \sigma)/\Sigma &= (\tau/\Sigma)(\sigma/\Sigma), \\
((\tau/\Sigma_1)/\Sigma_2 &= \tau/(\Sigma_1 \cap \Sigma_2)
\end{align*}
\]

for all traces \( \tau, \sigma \) over \( \Sigma \).

3. TRACE SYSTEMS

3.1. Trace languages. Let \( \Sigma \) be a concurrent alphabet, \( A \) be the alphabet of \( \Sigma \). Each subset of \( \Theta(\Sigma) \) is called a trace language over \( \Sigma \). Concatenation of trace languages is defined in usual way; for each trace languages \( T, S \), their concatenation is the language

\[
TS = \{\tau_1 \tau_2 \mid \tau_1 \in T, \tau_2 \in S\}.
\]

For each nonnegative integer \( n \) the power \( T^n \) of a trace language \( T \) over \( \Sigma \) is defined recursively:
\[ T^0 = \{\varepsilon\} \Sigma, \]
\[ T^{n+1} = T^nT, \quad n \in \omega. \]

The iteration of \( T \) is the union

\[ T^* = \bigcup_{n \in \omega} T^n. \]

The following formulas are valid for arbitrary (string) languages over \( \Sigma \), \( \mathbf{L}, \mathbf{L}_1, \mathbf{L}_2, \) and an arbitrary family \( \{\mathbf{L}_i\}_{i \in J} \) of such languages:

\[
\begin{align*}
\emptyset &= \emptyset, \\
[\mathbf{L}_1][\mathbf{L}_2] &= [\mathbf{L}_1\mathbf{L}_2], \\
\mathbf{L}_1 \subseteq \mathbf{L}_2 &\implies [\mathbf{L}_1] \subseteq [\mathbf{L}_2], \\
[\mathbf{L}_1] \cup [\mathbf{L}_2] &= [\mathbf{L}_1 \cup \mathbf{L}_2], \\
\bigcup_{i \in J} [\mathbf{L}_i] &= \bigcup_{i \in J} \mathbf{L}_i, \\
[\mathbf{L}]^* &= [\mathbf{L}^*].
\end{align*}
\]

For each trace language \( T \) define

\[ \text{Pref}(T) = \bigcup_{\tau \in T} \text{Pref}(\tau). \]

A trace language \( T \) is **prefix closed**, if \( T = \text{Pref}(T) \); in other words, \( T \) is prefix closed, if

\[ \tau \in T, \sigma \subseteq \tau \implies \sigma \in T. \]

A trace language \( T \) is **directed**, if

\[ \tau_1 \in T, \tau_2 \in T \implies \exists \tau \in T: \tau_1 \subseteq \tau, \tau_2 \subseteq \tau. \]

3.2. **Generalized dependency graphs.** Similarly as infinite strings can be defined as directed and prefix closed string languages (consisting of all finite prefixes of such an infinite string), infinite traces can be defined as directed and prefix closed trace languages; an infinite trace can be identified with a trace language consisting of all finite trace prefixes of such a trace. The notion of dependency graph can be easily generalized to this case. To do it, we need some auxiliary definitions. Let \( \Sigma \) be a concurrent alphabet; we shall consider oriented and acyclic graphs with nodes labelled with symbols from \( \Sigma \). A subgraph \( G_0 \) of such a graph \( G \) is the initial part of
G, if, together with a node v, \( B_0 \) contains all arcs in G leading to v. The class of all initial parts of a graph G will be denoted by Init(G).

THEOREM 4. For each directed and prefix closed trace language T over \( \Sigma \) there exists unique (up to isomorphism) oriented and acyclic graph G with nodes labelled with symbols in \( \Sigma \) such that for each \( \gamma \)

\[ \exists ! \tau \in T : \delta(\tau) \cong \gamma \iff \exists g \in \text{Init}(G) : \eta \cong \gamma \]

for each \( \gamma \) (\( \cong \) denotes the isomorphism of oriented labelled graphs). \( \Box \)

For each directed trace language T over \( \Sigma \) denote by \( \delta(T) \) and call the d-graph of T the graph the existence of which is ensured by Theorem 4. Clearly, \( \delta(T) \) need not be finite. Dependency graphs of directed trace languages are intended to describe explicitly the ordering of symbols (elementary actions) in processes represented by such languages.

EXAMPLE 9. Let \( \Sigma = \text{D}(a,b) \cup \text{D}(a,c) \). The trace language \( T = [(abc)^*] \) over \( \Sigma \) is directed; \( \delta(T) \) is the (infinite) graph given in Fig. 6 (as usual, arcs following by transitivity from others are omitted). \( \Box \)

![Diagram](image)

Fig. 6: An infinite dependency graph

3.3. Regular trace languages. Let \( T \subseteq \Theta(\Sigma) \); the decomposition of T is defined as the set

\[ \Delta_{\Sigma}(T) = \bigcup_{\tau \in T} \Delta_{\Sigma}(\tau). \]

We write \( \Delta(T) \) instead of \( \Delta_{\Sigma}(T) \), if \( \Sigma \) is understood. From the above definition it follows that \( \Delta(T) \) contains connected traces only. Furthermore, operation \( \Delta \) has the
following properties:

\[ T^* \subseteq \Delta(T)^*; \]
\[ T = \Delta(T), \text{ if the dependency in } \Sigma \text{ is full}; \]
\[ \Delta(\Delta(T)) = \Delta(T). \]

We say that a trace language \( T \) is connected, if \( T = \Delta(T) \). Obviously, if \( T, S \) are connected trace languages, then \( T \cup S \) is also connected.

**EXAMPLE 10.** Let \( \Sigma = D(a) \cup D(b) \). Then the language \( \{ab\} \) is not connected, since \( [a] \in \Delta(\{ab\}) = \{[a], [b]\} \) and \( [a] \notin \{ab\} \); the language \( \{a \cup b\} \) is connected. \( \square \)

The **trace iteration** \( T^\# \) of a trace language \( T \) is defined as

\[ T^\# = (\Delta(T))^*. \]

Roughly speaking, in \( T^\# \) connected components of a trace in \( T \) are iterated independently of each other (being treated as separate elements of \( T \)), in contrast to the ordinary iteration \( T^* \), where all components of a trace in \( T \) are iterated simultaneously. If dependency is full, then clearly \( T^* = T^\# \).

**EXAMPLE 11.** Let \( \Sigma = D(a) \cup D(b) \). Then \( \{ab\}^* \neq \{ab\}^\# \), since \( \{ab\}^* = \{[a^n b^n] \mid n \in \omega\} \), while \( \{ab\}^\# = \{[a^n b^m] \mid n, m \in \omega\} = \{a^* b^*\} \). \( \square \)

Let \( T \) be a trace language over a concurrent alphabet \( \Sigma \). For each trace \( \tau \) over \( \Sigma \) denote by \( (T/\tau) \) the set

\[ \{\sigma \in \Theta(\Sigma) \mid \tau \sigma \in T\}. \]

Similarly to the case of string languages, call a trace language \( T \) **regular**, if the family of sets

\[ (T/\tau) \mid \tau \in \Theta(\Sigma) \]

is finite. The following theorem characterizes regular trace languages in a similar way to the characterization of regular string languages; in fact, the following theorem is a generalization of the Kleene characterization of regular string languages.

**THEOREM 5** (Ochmański 1985). The class of all regular trace languages over \( \Sigma \) is the
least class of languages containing all finite trace languages over $\Sigma$ and closed w.r. to the union, concatenation, and the trace iteration operations. $\square$

Observe that in case of full dependency, i.e. when traces are isomorphic with strings, this theorem is reduced to the Kleene theorem, since in this case trace and string iterations are identical (up to isomorphism).

3.4. Fixed-point equations. Let $\Sigma_1 = (A, D_1)$, $\Sigma_2 = (B, D_2)$ be concurrent alphabets and let $\psi$ be a function:

$$\psi: 2^{A^*} \longrightarrow 2^{B^*}.$$ 

We say that $\psi$ is monotonic, if for each $X, Y \subseteq A^*$

$$X \subseteq Y \Rightarrow \psi(X) \subseteq \psi(Y),$$

and that $\psi$ is congruent (w.r. to $\Sigma_1, \Sigma_2$), if for each $X, Y \subseteq A^*$

$$[X]_{\Sigma_1} = [Y]_{\Sigma_1} \Rightarrow [\psi(X)]_{\Sigma_2} = [\psi(Y)]_{\Sigma_2}.$$ 

If $\psi$ is congruent, then there exists a function $\psi$,

$$\psi: 2^{\Theta(\Sigma_1)} \longrightarrow 2^{\Theta(\Sigma_2)}$$

such that for each $X \subseteq A^*$

$$\psi([X]_{\Sigma_1}) = [\psi(X)]_{\Sigma_2}.$$ 

Clearly, arbitrary superpositions of monotonic and congruent functions are monotonic and congruent. For any set $X$ and any function

$$\psi: 2^X \longrightarrow 2^X,$$

a subset $X_0$ of $X$ is the least fixed point of $\psi$, if

$$\psi(X_0) = X_0,$$

and for each $Y \subseteq X$
\[ \varphi(Y) = Y \Rightarrow \chi_0 \subseteq Y. \]

The following theorem expresses the relationship between solving equations (finding fixed points) in the domain of string languages and that of trace languages.

**THEOREM 6.** Let \( \Sigma = (A,D) \) be a concurrent alphabet. If

\[ \varphi : 2^A^* \longrightarrow 2^A^* \]

is monotonic and congruent, \( \chi_0 \) is the least fixed point of \( \varphi \), then [\( \chi_0 \)]\( \Sigma \) is the least fixed point of function [\( \varphi \)]\( \Sigma \) defined for all \( X \subseteq A^* \) by the equality

\[ [\varphi]_\Sigma ([X]_\Sigma) = [\varphi(X)]_\Sigma. \]

This theorem can be easily generalized for tuples of monotonic and congruent functions. As examples of monotonic and congruent functions can serve functions built up from variables and constants by means of union, concatenation and iteration operations. Another example of congruent functions are trace projections.

### 3.5. Trace systems

By a **trace system** we shall understand any ordered pair \((\Sigma,T)\), where \( \Sigma \) is a concurrent alphabet and \( T \subseteq \Theta(\Sigma) \). The class of all trace systems will be denoted by \( Z \). Similarly to string systems, let for each trace system \( X = (\Sigma,T) \)

\[ \alpha(X) = \Sigma, \lambda(X) = T. \]

We shall write \( \emptyset \) for the empty trace system \((\emptyset,();\emptyset))\) and \( X \subseteq Y \), if \( \alpha(X) \subseteq \alpha(Y) \) and \( \lambda(X) \subseteq \lambda(Y) \). Clearly, \( Z \) is partially ordered by \( \subseteq \). A trace system \( X \) is said to be: **sequential**, if \( \alpha(X) = D(A) \) for some alphabet \( A \), i.e. if the dependency in \( \alpha(X) \) is full; **regular** (directed, prefix closed), if \( \lambda(X) \) is a regular (directed, prefix closed, resp.) trace language over \( \alpha(X) \). If \( T \) is a trace in \( \Theta(\alpha(X)) \), define

\[ \tau X = (\alpha(X), \tau \lambda(X)) \]

and extend this definition for trace languages:

\[ TX = (\alpha(X), T \lambda(X)) \]

for each trace language \( T \) over \( \alpha(X) \).

The motivation for introducing trace systems is the same as that for string systems:
within trace systems we can speak not only about occurrences of symbols in traces, but also about their absence.

3.6. Synchronization. The synchronization operation is a trace analogue of the merging operation for string systems. By synchronization we mean a binary operation in $\Pi$, i.e. a function

$$\Pi: \Pi \times \Pi \rightarrow \Pi$$

such that for each trace systems $X, Y$

$$\alpha(X \Pi Y) = \alpha(X) \cup \alpha(Y),$$
$$\lambda(X \Pi Y) = \{\tau \in \Theta(\alpha(X \Pi Y)) \mid (\tau/\alpha(X)) \in \lambda(X), (\tau/\alpha(Y)) \in \lambda(Y)\}.$$ 

The infix notation is used here for $\Pi$ operation; $\Pi$ is assumed to bind weaker than the concatenation and stronger than other set-theoretical operations. Let $X, Y, Z$ be trace systems, $\Sigma_1, \Sigma_2$ be concurrent alphabets, $\tau$ be a trace over $\alpha(X) \cup \alpha(Y)$. The following equalities hold:

$$X \Pi X = X,$$
$$X \Pi Y = Y \Pi X,$$
$$X \Pi (Y \Pi Z) = (X \Pi Y) \Pi Z$$
$$([\varepsilon], \Sigma_1) \Pi ([\varepsilon], \Sigma_2) = ([\varepsilon], \Sigma_1 \cup \Sigma_2),$$
$$X \Pi \emptyset = \emptyset,$$
$$(X \cup Y) \Pi Z = X \Pi Z \cup Y \Pi Z,$$
$$\tau(X \Pi Y) = (\tau/\alpha(X)) \Pi (\tau/\alpha(Y)) Y,$$
$$(X \Pi Y) \tau = X(\tau/\alpha(X)) \Pi Y(\tau/\alpha(Y)).$$

The essential difference between the (string) merging $\&$ and the (trace) synchronization $\Pi$ is that the last operation applied to singletons returns either a singleton or the empty set, which (in general) is not true for the merging operation; it means that $\Pi$ restricted to singletons is a partial function, while $\&$ similarly restricted is not.

The following propositions express some preservation properties of the synchronization operation.

**Proposition 4.** If $X, Y$ are regular trace systems, then so is $X \Pi Y$. $\Box$
PROPOSITION 5. If \( X \), \( Y \) are prefix closed trace systems, then so is \( X \uparrow Y \). \( \square \)

The next theorem enables us to find the synchronization of trace languages by solving fixed point equations.

THEOREM 7. The synchronization operation \( \uparrow \) is the least binary operation in \( Z \) (w.r. to the inclusion ordering of \( Z \)) meeting the following properties:

1. \((\epsilon, A) \uparrow (\epsilon, B) = (\epsilon, A \cup B)\)
2. \((X \cup Y) \uparrow Z = X \uparrow Z \cup Y \uparrow Z\)
3. \(\tau(X \uparrow Y) = (\tau \circ \& (X)) \uparrow (\tau \circ \& (Y)) Y\)
4. \(X \uparrow Y = Y \uparrow X\)

for any concurrent alphabets \( A, B \), any trace systems \( X, Y, Z \), and each trace \( \tau \) in \( \Theta (\& (X) \cup \& (Y)) \). \( \square \)

EXAMPLE 12. Let \( A = D(a, b) \), \( B = D(a, c, d) \) and let \( X = (A, [(a b)^* (a \cup \epsilon)]) \), \( Y = (B, [(a c)^* (a d \cup a \cup \epsilon)]) \) be trace systems. Find \( X \uparrow Y \).

First, put \( X_0 = (A, [\epsilon]) \), \( Y_0 = (B, [\epsilon]) \), and \( Z_0 = (A \cup B, [\epsilon]) \). Next, write equations for \( X \) and \( Y \):

\[
X = [ab]X \cup ([a] \cup \{\epsilon\})X_0,
Y = [ac]Y \cup ([ad] \cup [a] \cup \{\epsilon\})Y_0.
\]

Hence,

\[
X \uparrow Y = ([ab]X \cup [a]X_0 \cup X_0) \uparrow ([ac]Y \cup [ad]Y_0 \cup [a]Y_0 \cup Y_0);
\]

by (i) we have

\[
X \uparrow Y = ([ab]X \uparrow [ac]Y) \cup ([a]X_0 \uparrow [ac]Y) \cup (X_0 \uparrow [ac]Y) \cup ([ab]X \uparrow [a]Y_0) \cup ([a]X_0 \uparrow [a]Y_0) \cup (X_0 \uparrow [a]Y_0) \cup ([ab]X \uparrow [ad]Y_0) \cup ([a]X_0 \uparrow [ad]Y_0) \cup (X_0 \uparrow [ad]Y_0) \cup ([ab]X \uparrow Y_0) \cup ([a]X_0 \uparrow Y_0) \cup (X_0 \uparrow Y_0);
\]

by (i) and (iii) we get

\[
[ab]X \uparrow [ac]Y = [abc](X \uparrow Y);\]
Thus, we have

\[ X\#Y = \{abc\}(X\#Y) \cup \{ac\}(X_0\#Y) \cup \{ab\}(X\#Y_0) \cup \{a\}(X_0\#Y_0) \cup \{abd\}(X\#Y_0) \cup \{ad\}(X\#Y_0). \]

Now we need to calculate \( X_0\#Y_0, X_0\#Y, \) and \( X\#Y_0 \):

\[ X_0\#Y_0 = Z_0, \quad \text{by (i)}, \]
\[ X_0\#Y = X_0\#(\{ac\}Y \cup \{a\}Y_0 \cup \{ad\}Y_0 \cup Y_0) = X_0\#Y_0 = Z_0, \]
\[ X\#Y_0 = \{ab\}X \cup \{a\}X_0 \cup X_0 \#Y_0 = X_0 \#Y_0 = Z_0. \]

Denote \( X\#Y \) by \( Z \); then \( Z \) satisfies the following equation:

\[ Z = \{abc\}Z \cup (\{ab\} \cup \{ac\} \cup \{abd\} \cup \{ad\} \cup \{a\} \cup \{e\})Z_0. \]

By Theorem 6 and Theorem 7 the least solution of the above equation is

\[ \{abc\}^*\{abd \cup ab \cup ad \cup ac \cup a \cup e\}Z_0 \]
\[ = \text{Pref}\{\{abc\}^*\{abd\}\}Z_0 \]
\[ = (A \cup B, \text{Pref}\{\{abc\}^*\{abd\}\}) \]
\[ = (A \cup B, \text{Pref}\{\{abc\}^*\{abd\}\}). \]

Since \( \{abc\} \) is a connected trace language over \( A \cup B \) we have as well

\[ X\#Y = (A \cup B, \text{Pref}\{\{abc\}^*\{abd\}\}) \]

which proves the regularity of \( X\#Y \). Observe that \( X \) and \( Y \) are sequential trace systems, but \( X\#Y \) is not. Since \( \alpha(X\#Y) = A \cup B = D(a,b) \cup D(a,c,d), (b,c) \) and \( (b,d) \) are independent in this system. The diagram in Fig. 6 represents the prefix ordering of \( X\#Y \). \( \square \)
Fig. 6: Prefix structure of Pref[(abc)*abd] with bc ≡ cb, bd ≡ db.
4. NETS AND TRACES

4.1. Elementary net systems. By an elementary net system (EN system, for short), we understand any ordered quadruple

\[ N = (B, E; F, C_{in}) , \]

where \( B \) and \( E \) are finite, disjoint, nonempty sets (of conditions and events of \( N \), respectively), \( F \subseteq B \times E \cup E \times B \) (the flow relation of \( N \)), with \( \text{dom}(F) \cup \text{cod}(F) = B \cup E \), and \( C_{in} \subseteq B \) (the initial case of \( N \)). As all other types of nets, EN systems are represented graphically using boxes to represent events, circles to represent conditions, and arrows leading from circles to boxes or from boxes to circles to represent the flow relation; in such a representation circles corresponding to conditions in the initial case are marked with dots.

EXAMPLE 13. The following quadruple is an EN system:

\[ N = ((1,2,3,4), \{a,b,c,d\}; \]
\[ \{(1,a),(a,3),(3,b),(b,1),(2,a),(a,4),(4,c),(4,d),(c,2),(1,2)\} . \]

In Fig. 7 this system is represented graphically. ☐

![Fig. 7: An elementary net system](attachment:image.png)

Let \( N = (B, E; F, C_{in}) \) be an EN system. For each \( x \in B \cup E \), the sets:
\[ x = \{ y \mid (y, x) \in F \}, \]
\[ x^* = \{ y \mid (x, y) \in F \}, \]
\[ x^* = x \cup x^* \],

are called respectively the \textbf{preset}, the \textbf{postset}, and the \textbf{proximity} of \( x \). Two EN systems are said to be \textbf{B-disjoint} (E-disjoint), if their sets of conditions (events, resp.) are disjoint.

\textbf{4.2. Firing sequences.} Let \( N = (B, E; F, C_1, n) \) be an EN system. The \textbf{transition function} \( \delta_N \) of \( N \) is a partial function:

\[ \delta_N : 2^B \times E \longrightarrow 2^B \]

such that

\[ \delta_N(C, e) = (C \setminus \cdot e) \cup e', \text{if } \cdot e \subseteq C, e' \cap C = \emptyset, \text{and undefined otherwise.} \]

Function \( \delta_N \) describes a change of conditions holding caused by a single event occurrence, if it is defined; otherwise the event has no concession to occur. The \textbf{reachability function} \( R_N \) of \( N \) is a partial function:

\[ R_N : E^* \longrightarrow 2^B, \]

such that

\[ R_N(\varepsilon) = C_1, \]
\[ R_N(we) = \delta_N(R_N(w), e), \]

for all \( e \in E, w \in E^* \). Set

\[ S_N = \text{Dom}(R_N), \ C_N = \text{Cod}(R_N), \]

and call elements of \( S_N \) (i.e. some sequences of events) \textbf{firing sequences} of \( N \), and those of \( C_N \) (i.e. some sets of conditions) \textbf{cases} of \( N \). If \( w \) is a firing sequence and \( R_N(w) = C \), then \( w \) is said to \textbf{lead} to the case \( C \) of \( N \).

The string system \((E, S_N)\) will be referred to as the \textbf{sequential behaviour} of \( N \) and denoted by \( \text{SB}_N \). Clearly, \( \text{SB}_N \) is prefix closed and ordered into a tree by the prefix
ordering. Maximal directed subsets of $S_N$ are then full paths through this tree and can be viewed as sequential observations of the behaviour of $N$, i.e. observations made by observers capable to see only a single event occurrence at a time. The ordering of symbols in strings of $SB_N$ reflects not only the (objective) causal ordering of event occurrences but also a (subjective) observational ordering resulting from a specific view over concurrent actions. Therefore, the structure of $SB_N$ alone does not allow to decide whether the difference in ordering is caused by a conflict resolution (a decision made in the system), or by different observations of concurrency. In order to extract from $SB_N$ the causal ordering of event occurrences we must supply $SB_N$ with additional information; as such information we take here the dependency of events.

4.3. Firing traces. Let $N = (B, E; F, C_1 \ldots n)$ be an EN system fixed for this section. Define dependency in $N$ as a relation $D_N \subseteq \text{EXE}$ such that:

$$(e_1, e_2) \in D_N \iff \text{`e}_1 \backslash \text{`e}_2 \neq \emptyset.$$ 

Clearly, $D_N$ is a reflexive and symmetric. The complement of $D_N$ in $E$ will be called the independency in $N$. By definition, two events are independent in $N$, if and only if they have no adjacent conditions in common; if such events occur next to each other in a firing sequence, the order of their occurrences is irrelevant, since (according to our understanding of EN system activity) they occur concurrently in this execution. In order to get rid of this ordering we intend to replace the sequential description of EN systems behaviour by nonsequential one, expressing the behaviour of EN systems by traces rather than sequences. To this end, set $\Sigma_N = (E, D_N)$ and call $\Sigma_N$ the concurrent alphabet associated with $N$; let $\equiv_N$ denote the trace equivalence over $\Sigma_N$ and $[w]_N$ the equivalence class of $\equiv_N$ represented by $w$. As usual, the subscript $N$ will be omitted if it causes no ambiguity.

The trace behaviour of $N$ is defined as a trace system $TB_N = (\Sigma_N, [S_N])$; elements of $TB_N$ will be called firing traces of $N$.

It is clear that each firing sequence of $N$ is contained in some equivalent class belonging to $TB_N$. Conversely, from the theorem below it follows that each representant of a firing trace of $N$ is a firing sequence of $N$ and all equivalent firing sequences lead to the same case of $N$.

THEOREM 8. The reachability function $R_N$ is congruent w.r. to $\Sigma_N$, i.e. for each two strings $w_1, w_2$ in $E^*$
\[ w_1 \equiv_N w_2 \Rightarrow R_N(w_1) = R_N(w_2) \]

(here, the equality in the conclusion means that either both sides are defined and
equal, or both of them are undefined). \(\Box\)

This is another motivation for introducing traces for describing concurrent systems
behaviour. Representing traces by their dependency graphs, firing traces can be viewed
as partially ordered sets of symbol occurrences; the ordering within firing traces is
determined by mutual dependencies of events and then it reflects the causal
relationship between event occurrences rather than the non-objective ordering
following from the string representation of the EN system activity.

**PROPOSITION 6.** \(TB_N\) is a prefix closed trace system. \(\Box\)

Let \(\Sigma\) be a concurrent alphabet. Two traces over \(\Sigma\) are said to be consistent, if both
of them are prefixes of a common trace over \(\Sigma\). A trace system \((\Sigma; X)\) is proper, if each
two consistent traces in \(X\) are prefixes of a common trace in \(X\). Levi Lemma for traces
and the transition function definition implies the following proposition.

**PROPOSITION 7.** \(TB_N\) is a proper trace system. \(\Box\)

Each firing trace uniquely determines a case of the EN system reached from its initial
case. Moreover, each firing trace determines uniquely the way of reaching this case
(the history of the EN system activity resulting with this case). Therefore, firing
traces can be identified with (global) states of an EN system. In this approach a
state defines, roughly speaking, what has already happened in the EN system. This
identification would be impossible while dealing with firing sequences, since some
different strings might represent the same history of the considered system; the one
to one correspondence between firing sequences and states would not be achieved before
identifying some of sequences into equivalence classes.

Directed subsystems of \(TB_N\) will be called (trace) processes in \(N\). If \(P\) is a process in
\(N\), then by its directedness and prefix closedness there exists a (generalized)
dependency graph of \(P\); thus, processes in \(N\) can be viewed as (possibly infinite)
dependency graphs over \(\Sigma_N\). The set of all processes in \(N\) describes all possible
executions of the system represented by \(N\).

4.4. Composition and decomposition of EN systems. In this section a modular way
of finding the behaviour of EN systems is presented. It will be shown how to construct
the behaviour of an EN system from behaviours of some parts of this EN system. The method is to decompose a given EN system into modules, to find their behaviours, to put them together, and to get in this way the behaviour of the original EN system.

We say that an EN system $N = (E, B; F, C)_{in}$ is composed of EN systems $N_1 = (E_1, B_1; F_1, (C_{in})_1)$, $N_2 = (E_2, B_2; F_2, (C_{in})_2)$, and write

$$N = N_1 + N_2,$$

if $N_1$, $N_2$ are B-disjoint (i.e. $B_1 \cap B_2 = \emptyset$), $E = E_1 \cup E_2$, $B = B_1 \cup B_2$, $F = F_1 \cup F_2$, and $C_{in} = (C_{in})_1 \cup (C_{in})_2$. Obviously, $(C_{in})_1 \cap (C_{in})_2 = \emptyset$. If $N = N_1 + N_2$, $N_1$ and $N_2$ are called components of $N$.

PROPOSITION 8. For each pairwise B-disjoint EN systems $N_1$, $N_2$, $N_3$:

$$N_1 + N_2 = N_2 + N_1,$$

$$(N_1 + N_2) + N_3 = N_1 + (N_2 + N_3). \Box$$

Thus, $+$ is a symmetric and associative operation on B-disjoint EN systems; it makes possible to omit parentheses and disregard the order of terms in composition.

PROPOSITION 9. For each B-disjoint EN systems $N_1$, $N_2$:

$$\Sigma_{N_1 + N_2} = \Sigma_{N_1} \cup \Sigma_{N_2}. \Box$$

The main theorem of this section is the following:

THEOREM 9. If $N_1$, $N_2$, ..., $N_k$ are pairwise B-disjoint EN systems, then

$$TB_{N_1 + N_2 + \ldots + N_k} = TB_{N_1} \| TB_{N_2} \| \ldots \| TB_{N_k}. \Box$$

This theorem allows us to find the behaviour of an EN system knowing behaviours of its components. One can expect the behaviour of components to be easier to compute than that of the whole EN system. In fact, there exists a standard set of very simple EN systems, with behaviours already known, such that an arbitrary EN system can be composed from systems in this set; these EN systems, containing only one condition, will be called atomic EN systems or simply atoms.

Let $N = (E, B; F, C_{in})$ be an EN system; for each $b \in B$ the EN system
\[ N_b = (b', b), F_b, (C_{in} b), \]

where

\[ F_b = \{(e, b) \mid e \in b'\} \cup \{(b, e) \mid e \in b'\}, \]

\[ (C_{in} b) = C_{in} \cap \{b\}, \]

is called an atom of \( N \) (determined by \( b \)). Clearly, the following proposition holds:

**Proposition 10.** Each EN system is the composition of all its atoms. \( \square \)

In Fig. 8 atoms of the EN system from Example 13 are presented.

![Atomic EN systems](image)

**Proposition 11.** Let \( N = (B, E; F, C_{in}) \) be an EN system, \( b \in B, U = b', V = b', A = b' \). Then

\[ TB_{N_b} = (D(A), ([((VU)^*(V \mathbin{\cup} \varepsilon)))), \quad \text{if } b \in C_{in} \text{ and} \]

\[ TB_{N_b} = (D(A), ([((UV)^*(U \mathbin{\uplus} \varepsilon)))), \quad \text{if } b \notin C_{in}. \]

This proposition together with Theorem 9 is to the effect that the activity of an arbitrary EN system can be found by performing the synchronization operation applied to trace behaviours of atoms of the system; these behaviours are given by the above expressions. Thus, for an arbitrary EN system it is possible to write down an explicit expression built up from (unique) event names and operations: \( \varepsilon \) (nullary), \( \ast \) (unary), and \( \mathord{\uplus}, \mathord{\uplus}, \mathord{\mathbb{1}} \) (binary).

**Example 14.** Consider EN system \( N \) from Example 13. Trace behaviours of its atoms (presented in Fig. 8) are:
\[ TB_{N_1} = (D(a, b), \{(ab)^*(a \lor \varepsilon)\}) \]
\[ TB_{N_2} = (D(a, c), \{(ac)^*(a \lor \varepsilon)\}) \]
\[ TB_{N_3} = (D(a, b), \{(ab)^*(a \lor \varepsilon)\}) \]
\[ TB_{N_4} = (D(a, c, d), \{(a(c \lor d))^*(a \lor \varepsilon)\}) \]

By idempotency of \( \# \) we get

\[ TB_{N_1} + N_2 = TB_{N_1} \# TB_{N_3} = (D(a, b), \{(ab)^*(a \lor \varepsilon)\}) \]

Now, set

\[ X = TB_{N_2}, \ Y = TB_{N_4}, \ X_0 = (D(a, c), \{\varepsilon\}), \ Y_0 = (D(a, b, c), \{\varepsilon\}) \]

by Theorem 6 we have the following equations for \( X \) and \( Y \):

\[ X = (ac)X \lor (a \lor \&)X_0, \ Y = (ac \lor \text{adj}Y) \lor (a \lor \varepsilon)Y_0 \]

and by Theorem 7 we get the following equation for \( XY \):

\[ XY = (ac)X \# (ac)Y \lor (ac)X \# (ac)Y \lor (ac)X \# (a \lor \varepsilon)Y_0 \lor (a \lor \varepsilon)X_0 \# (a \lor \varepsilon)Y_0 \]

\[ = (ac)X \# (ac)Y \lor (a)X_0 \# (a)Y_0 \]

(all remaining terms vanish); by Theorem 7(111) we obtain

\[ XY = (ac)(XY) \lor (ad)(X_0 \# Y_0) \lor (a)(X_0 \# Y_0) \lor (\varepsilon)(X_0 \# Y_0) \]

set \( Z = XY \), \( Z_0 = X_0 \# Y_0 \); since \( X_0 \# Y = (D(a, c, d), \{\varepsilon\}) = Z_0 \) we have the following equation:

\[ Z = (ac)Z \lor (ad \lor a \lor \varepsilon)Z_0 \]

for \( XY \). By Theorem 6 the least solution of this equation is \[ (ac)^*(ad \lor a \lor \varepsilon)Z_0 \]; thus,

\[ TB_{N_2} + N_4 = (D(a, c, d), \{(ac)^*(ad \lor a \lor \varepsilon)\}) \]

To get \( TB_N \) we need to calculate
\[ TB_{N_1} = (D(a,b),[(ab)^ *(a \lor e)]), \]
\[ TB_{N_2} = (D(a,c),[(ac)^ *(a \lor e)]), \]
\[ TB_{N_3} = (D(a,b),[(ab)^ *(a \lor e)]), \]
\[ TB_{N_4} = (D(a,c,d),[(a(c \lor d)^ *(a \lor e)])]. \]

By idempotency of \( \# \) we get
\[ TB_{N_1} \# TB_{N_3} = TB_{N_1}\]
\[ TB_{N_1} \# TB_{N_3} = (D(a,b),[(ab)^ *(a \lor e)]). \]

Now, set
\[ X = TB_{N_2}, \quad Y = TB_{N_4}, \quad X_0 = (D(a,c),[e]), \quad Y_0 = (D(a,b,c),[e]); \]
by Theorem 6 we have the following equations for \( X \) and \( Y \):
\[ X = [(ac)X \lor (a \lor \&)X_0, \quad Y = [(ac \lor ad)Y \lor (a \lor e)Y_0; \]
and by Theorem 7 we get the following equation for \( X \# Y \):
\[ X \# Y = [(ac)X \# (ac)Y \lor (ac)X \# (ad)Y \lor (ac)X \# (a \lor e)Y_0 \lor \]
\[ (a \lor e)X_0 \# (ac \lor ad)Y \lor (a \lor e)X_0 \# (a \lor e)Y_0 \]
\[ = [(ac)X \# (ac)Y \lor (a)X_0 \# (ad)Y \lor (a)X_0 \# (a)Y_0 \lor (e)X_0 \# (e)Y_0] \]
(all remaining terms vanish); by Theorem 7(iii) we obtain
\[ X \# Y = [(ac)(X \# Y) \lor (ad)(X_0 \# Y_0) \lor (a)(X_0 \# Y_0) \lor (e)(X_0 \# Y_0)]; \]
set \( Z = X \# Y, \quad Z_0 = X_0 \# Y_0 \); since \( X_0 \# Y = (D(a,c,d),[e]) = Z_0 \) we have the following equation:
\[ Z = [(ac)Z \lor (ad)Z \lor (a)Z \lor (e)Z]_0 \]
for \( X \# Y \). By Theorem 6 the least solution of this equation is \([(ac)^ *(ad \lor a \lor e)]Z_0; \)
thus,
\[ TB_{N_2} \# N_4 = (D(a,c,d),[(ac)^ *(ad \lor a \lor e)]). \]

To get \( TB_N \) we need to calculate
\[ \sigma_1 \in \sigma_2 \iff \sigma_1 \in \text{Pref}(\sigma_2). \]

For any subset \( Q \) of \( \Theta \) define

\[ I_Q = \{ \tau \in \Theta \mid \exists \sigma \in Q : \sigma \subseteq \tau \}. \]

\[ I_Q = \{ \tau \in \Theta \mid \exists \sigma \in Q : \tau \subseteq \sigma \}. \]

We say that a subset \( Q \) of \( \Theta \) dominates another subset \( R \) of \( \Theta \) (or that \( R \) is dominated by \( Q \)), if \( R \subseteq I_Q \). Two states are consistent, if there is a state in the state space dominating both of them and inconsistent otherwise. A set \( P \) of states is said to be proper, if any two of its consistent states are dominated by a state in \( P \), and directed, if arbitrary two states in \( P \) are dominated by a state in \( P \). Clearly, any directed set of states is proper, but not the other way around. A maximal chain in \( P \) will be called a line in \( P \). A set of states is conflict-free, if any two of its states are consistent, and it is sequential, if any two of its consistent states are comparable. Thus, each sequential set of states is proper.

5.2. Trace structures. By a trace structure over \( \Sigma \) we shall mean any prefix closed and proper trace language over \( \Sigma \); traces in a trace structure \( T \) will be called states of \( T \).

From Propositions 14 and 15 it follows that if \( (\Sigma, T) \) is the trace behaviour of any EN system, \( T \) is a trace structure over \( \Sigma \); the converse is not true in general. By the diagram of a trace structure \( T \) we shall understand the graph of the poset \( (T, \subseteq) \); in such a graph arcs resulting from others by transitivity or reflexivity will be omitted. Examples of such graphs were given in Fig. 5(b) and Fig. 6.

Trace structures are intended to represent the behaviour of concurrent systems. Let \( T \) be a trace structure and let \( \sigma', \sigma'' \) be states in \( T \). As it has been mentioned above, if a state \( \sigma' \) is dominated by another state \( \sigma'' \), it means that there is a run of the represented concurrent system, containing both of them, in which \( \sigma'' \) comes up not earlier than \( \sigma' \). Thus, two states are comparable, if and only if they appear in the same execution of the concurrent system and the history of one of them is an initial part of the history of the other. If \( \sigma' \) is incomparable with \( \sigma'' \), two cases are possible: either both of them are dominated by a common state (then they are consistent), or such a common state does not exist (then they are inconsistent). In the first case \( \sigma' \) and \( \sigma'' \) are two states of the same run of the concurrent system, but resulting in effect of concurrent actions of the system; they identify different
pieces of the same history, exhibited later by the common dominating state. In the second case \( \sigma' \) and \( \sigma'' \) are states identifying pieces of two different histories, resulting in effect of a choice (a conflict resolution) made earlier during the system action.

A trace structure \( T \) is **sequential (conflict free)**, if \( T \) is a sequential (conflict free) trace language.

**PROPOSITION 12.** A trace structure is sequential if and only if its diagram is a rooted tree; a trace structure is conflict free, if and only if it is directed. \( \Box \)

Maximal (w. r. to the inclusion ordering) directed subsets of a trace structure \( T \) will be called **processes** in \( T \). Clearly, all processes are prefix closed. Because of directedness, any process contains only states appearing in a single run of the represented system, since all of them can be viewed as prefixes of the same history. Because of maximality, processes describe full executions of the concurrent system being interpreted.

**EXAMPLE 15.** It has been shown that the trace behaviour of the EN system from Example 13 is \((\Sigma, T)\) where

\[
\Sigma = D(a,b) \cup D(a,c,d), \quad T = \text{Pref}(\langle abc \rangle^\ast \text{abd});
\]

\( T \) is a trace structure; the diagram of \( T \) has been presented in Fig. 6. Processes in \( T \) are languages

\[
P_0, P_1, \ldots, P_n, \ldots, P_\omega,
\]

where \( P_n = \text{Pref}(\langle abc \rangle^n \text{abd}) \) for each \( n \in \omega \), and \( P_\omega = \text{Pref}(\langle abc \rangle^\ast) \). Observe that for each \( n \) process \( P_n \) is finite, and \( P_\omega \) is infinite.

Diagrams of \( P_n \) and of \( P_\omega \) are given in Fig. 6.

**PROPOSITION 13.** The set of processes of any trace structure is not empty. \( \Box \)

Let \( T \) be a trace structure; a state is **terminal** in \( T \), if it is a maximal state of \( T \); clearly, if a trace structure \( T \) contains a terminal state \( \sigma \), then \( \text{Pref}(\sigma) \) is a finite process in \( T \); conversely, any finite process in \( T \) is equal to \( \text{Pref}(\sigma) \) for some terminal state \( \sigma \).
PROPOSITION 14. The only process in a conflict free trace structure is the trace structure itself. □

PROPOSITION 15. A subset of a sequential trace structure $T$ is a process in $T$ if and only if it is a line in $T$. □

Fig. 7: Processes $P_n (a)$ and $P_w (b)$

5.3. Observations. To prove inevitable properties of concurrent systems (discussed in the next section) we need the notion of an observation. Intuitively speaking, an observation is a finite or infinite sequence of states as seen by a sequential observer of a system. Because of possible concurrency, different observers can notice different order of action execution and, consequently, their observations consist of different system states. Objective properties of the system behaviour are independent on particular observations, hence such properties should be discovered by all system observations provided they are faithful, i.e. they do not overlook any action performed by the system. These intuitions give rise to the following formal definition.
By an observation of a trace structure \( T \) we mean any line \( V \) in \( T \) such that \( \downarrow V \) is a process in \( T \).

Recall here that any process is a maximal directed subsystem of \( T \), i.e. it contains every possible state of a single run of the represented system; each line is a maximal sequential observation of the system run; to guarantee the "faithfulness" of such an observation we require the maximality of \( \downarrow V \). The set \( \downarrow V \) of states is obviously directed for any line \( V \) in \( T \), but not necessarily it is a maximal directed subsystem of \( T \). Because of this maximality property, one can say that an observation is a line that dominates everything possible to be dominate by a line.

**Theorem B.** For each system \( T \) and each process \( P \) in \( T \) there exists an observation of \( T \) dominating \( P \). \( \Box \)

**Example 16.** Let \( E = D(a) \cup D(b) \), \( T = \{a^i b^j \mid 0 \leq j \leq i < \omega \} \). \( T \) is clearly a system. The diagram of \( T \) is given in Fig. 7.

![Diagram of system T](image)

Fig. 7

Observe first that the system is conflict free (all its states are consistent); thus, any observation must dominate the whole system. The line

\[ V = \{ \varepsilon \}, \{a\}, \{ab\}, \{aba\}, \{abab\}, \ldots, \{(ab)^n\}, \{(ab)^n a\}, \ldots \]

is an observation of \( T \), since any state in \( T \) is dominated by a state in \( V \), e.g.

\[ [abbaabaaa] = [aaaaaabbbb] \in [aaaaaabbbb] = [ababababab] \in V. \] However, not all
lines in $T$ are observations; e.g. the line

$$U = \{[e],[a],[aa],...,[a^n],...\},$$

is not an observation, since it leaves undominated all states $[a^ib^j]$ with $j > 0$.

Observe that in $T$ there is infinitely many lines that are observations of $T$ as well as infinitely many lines that are not such observations. Observe also that each finite chain in $T$ can be extended to an observation of $T$ as well as to a line being not an observation of $T$. It means that the property of a line "to be an observation" is not finitary; it is not sufficient to know a finite part of a line to decide whether the line is an observation or not. □

Call a trace structure $T$ well synchronized, if for each state $\sigma$ of $T$ there exists a finite number of states of $T$ incomparable and consistent with $\sigma$.

PROPOSITION 16. Every line in a well synchronized trace structure is an observation. □

EXAMPLE 17. The system presented in Fig. 6 is well synchronized; consequently, every line in it is an observation. The system presented in Fig. 7 is not well synchronized; for each $n \in \omega$ state $[a^{n+2}]$ is consistent and incomparable with the state $[ab]$. □

5.4. Invariancy and inevitability. Let $T$ be a trace structure over $\Sigma$; by a property we shall mean any subset of states in $\Theta(\Sigma)$. We say that a property $\Theta$ is invariant in $T$, if and only if

$$\lambda(T) \wedge \Theta \subseteq \Theta.$$ □

It means that once the system reaches a state with an invariant property, all subsequent states will have this property.

THEOREM 12. A property $\Theta$ is invariant in a trace structure $T$ over $(A,D)$, if for each $\sigma \in \Theta$ and each $a \in A$

$$\sigma[a] \in T \Rightarrow \sigma[a] \in \Theta.$$ □

Thus, checking the invariancy of a property in concurrent systems is relatively simple and similar to that in sequential ones: $\Theta$ is invariant, if for each state in $\Theta$ any possible next state is also in $\Theta$. 
PROPOSITION 15. \( \emptyset \) is an invariant property; \( T \) is an invariant property; if \( \Theta_1 \) and \( \Theta_2 \) are invariant properties, then

\[
\Theta_1 \cup \Theta_2, \quad \Theta_1 \cap \Theta_2
\]

are also invariant properties. \( \square \)

In order to define inevitability in trace structures we have to refer to observations. Intuitively, a property is inevitable in a system, if sooner or later the system reaches a state with this property. Inevitability of a property must be objective, i.e. it must not depend on particular observations of the system; hence, if a property is inevitable, any observer of the system must sooner or later notice a state with this property. The formal definition is as follows.

A property \( \Theta \) is inevitable in \( T \), if each observation of \( T \) intersects \( \Theta \) (i.e. if \( V \cap \Theta \neq \emptyset \) for any observation \( V \) of \( T \)).

A property that is not inevitable is called avoidable.

EXAMPLE 18. Consider system \( T \) from Example 17. The (one state) property \( \Theta_{nm} \):

\[
\Theta_{nm} = \langle [a^n b^m] \rangle
\]

is avoidable in \( T \) for any \( n, m \in \omega \), since the following line is an observation of \( T \) not intersecting \( \Theta_{nm} \):

\[
[\varepsilon], [a], [a^n], [a^{n+1}], [a^{n+1} b], \ldots, [a^{n+1} b^m], [a^{n+1} b^m a], \ldots, [a^{n+1} b^m (ab)^n], \ldots
\]

The property \( R_{nm} \):

\[
R_{nm} = \langle [a^i b^j] \mid i \leq n, \ j \leq m \rangle
\]

is inevitable in \( T \) for each \( n, m \in \omega \), since any observation of \( T \) must contain a state dominating \([a^k b^k]\) with \( k = \max(n, m)\); such a state clearly belongs to \( R \). \( \square \)

In general, it is more difficult to prove inevitability than invariancy, mainly because of the nonlocal character of the observation definition. The following proposition gives simple facts about inevitability that can be useful in checking inevitability of some properties.
PROPOSITION 15. T is inevitable in T; (ε) is inevitable in T; if Q₁ ≤ Q₂ and Q₁ is inevitable in T, then Q₂ is inevitable in T; if each process in T contains a state in 0, then 10 is inevitable in T. Q

Proving invariance and/or inevitability of some properties (and some combination of them) is of primary interest in analysis and design of concurrent systems. The above formulated notions are thought as some "trace" primitives which can be helpful in creating a partial order model for developing a temporal logic with concurrency.

6. BIBLIOGRAPHICAL NOTES

The motivation for developing the presented theory was to create means for analysing the behaviour of Petri nets as introduced in [Pet1-Pet2], [R1], [RT1]. The notion of traces and of dependency graphs has been introduced in [M1] and presented at Aarhus Workshop on Concurrency in 1977. Since then the theory started to develop in several directions: investigation of algebraic properties [AW1], [BBMS1], [BG1], [BMS1-BMS7], [Kn1-Kn2], [KB], [M4], [Sz1], [T1], [O1], [S1], [Ry1]; connections with graph theory [AR1]; applications to the analysis of concurrent systems [FR], [J1-J4], [Z1-Z2], [Z1]; comparison with other formalisms [KB1], [M4]. Independently, since 1978 the algebra of free partially commutative monoids started to develop [[CdL1], [CF1], [CL1-CL2], [CM1], [CF1], [D1-D2], [F1], [Me1-Me2], [P1]. Since the application of traces is limited to finite (0,1)-marked nets, some more sophisticated tools has been proposed in [G1], [St1-St2]. Traces introduced in [Re1] correspond to traces presented here but without independency of events.

Elementary net systems considered here has been introduced in [Ro1], [Th1]. In [Ro1] an application of traces for the behaviour description is given. The synchronization operation for trace systems has been defined in [M5] with an extensive use of projections due to [GKR1], [KS1]. Modular and fixed point technique for calculating the trace behaviour of nets is given in [M5]. Trace structures are similar to the event structures introduced in [NPW1], but based on dependency / independency relation rather than the conflict relation. The notion of inevitability in the trace structure framework is a particular case of a more general concept discussed in a forthcoming paper.

Acknowledgements. The author gratefully acknowledges the help of G. Rozenberg in
preparation of this paper. Thanks are due to my colleagues E. Ochmański and W. Penczek who were the first readers of the paper. The author is indebted to IJ. J. Aalbersberg for his effort in collecting the bibliography on traces.

7. REFERENCES


[AR2]: AALBERSBERG, IJ. J., ROZENBERG, G.: "Trace languages defined by context-free languages", manuscript, (1985)

[AW1]: AALBERSBERG, IJ. J., WELZL, E.: "Trace languages defined by regular string languages", to appear (1985)


[BMS1]: BERTONI, A., MAURI, G., SABADINI, N.: "A hierarchy of regular trace languages and some combinatorial applications", Proc. 2nd World Conf. on Math. at the Serv. of Men, Las Palmas (1982) 146 - 153


[BMS4]: BERTONI, A., MAURI, G., SABADINI, N.: "Equivalence and membership problems for regular and context - free trace languages", manuscript (1982)


[BMS7]: BERTONI, A., MAURI, G., SABADINI, N.: "Representation of prefixes of a trace and membership problem for context - free trace languages", manuscript (1985)

[BFP1]: BEST, E., FERNANDEZ, C. and PLUNNECKE, H.: "Concurrent systems and processes", manuscript (1985)


[F1]: FOATA, D.: "Rearrangements of words", in M. Lothaire: Combinatorics on words, Addison-Wesley, Reading, (1983) Chapter 10


[Kn1]: KNUTH, E.: "Petri nets and regular trace languages", manuscript (1978)


[KB1]: KNUTH, E. and GYORY, G.: "Paths and traces", Computational Linguistics and
LEVI, F. W.: "On semigroups", Bull. of the Calcutta Math. Soc. 36 (1944) 141 - 146


METIVIER, Y.: "Sous-ensemble reconnaissable d'un monoïde partiellement commutatif libre", manuscript (1985)


ROZENBERG, G.: "Behaviour of elementary net systems", this volume (1986)


[St2]: STARKE, P. H.: "Traces and semiwords", Lecture Notes in Computer Science 208 (1985) 332 - 349


[Th1]: THIAGARAJAN, P.S.: "Elementary net systems", this volume (1986)


[Zu1]: ZUIDWEG, H.: "Trace approach to synchronic distances" (manuscript), University of Leiden, Leiden (1986)