COMPUTING
SCIENCE

Hasse Diagrams of Combined Traces

Lukasz Mikulski and Maciej Koutny
Hasse Diagrams of Combined Traces

L. Mikulski, M. Koutny

Abstract

One of the standard ways to represent concurrent behaviours is to use concepts originating from language theory, such as traces and comtraces. Traces can express notions such as concurrency and causality, whereas comtraces can also capture weak causality and simultaneity. This paper is concerned with the development of efficient data structures and algorithms for manipulating comtraces. We introduce Hasse diagrams for comtraces which are a generalisation of Hasse diagrams defined for partial orders and traces, and develop an efficient algorithm for deriving them from language theoretic representations of comtraces. We also explain how the new representation of comtraces can be used to implement efficiently some basic operations on comtraces.
Bibliographical details

MIKULSKI, L., KOUTNY, M.
Hasse Diagrams of Combined Traces
[By] L. Mikulski, M. Koutny
(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1301)

Added entries

NEWCASTLE UNIVERSITY

Abstract

One of the standard ways to represent concurrent behaviours is to use concepts originating from language theory, such as traces and comtraces. Traces can express notions such as concurrency and causality, whereas comtraces can also capture weak causality and simultaneity. This paper is concerned with the development of efficient data structures and algorithms for manipulating comtraces. We introduce Hasse diagrams for comtraces which are a generalisation of Hasse diagrams defined for partial orders and traces, and develop an efficient algorithm for deriving them from language theoretic representations of comtraces. We also explain how the new representation of comtraces can be used to implement efficiently some basic operations on comtraces.

About the authors

Łukasz Mikulski is a Junior Lecturer at the Faculty of Mathematics and Computer Science of the Nicolaus Copernicus University, Torun, Poland.

Maciej Koutny obtained his MSc (1982) and PhD (1984) from the Warsaw University of Technology. In 1985 he joined the then Computing Laboratory of the University of Newcastle upon Tyne to work as a Research Associate. In 1986 he became a Lecturer in Computing Science at Newcastle, and in 1994 was promoted to an established Readership at Newcastle. In 2000 he became a Professor of Computing Science.

Suggested keywords

COMTRACE
TRACE
CONCURRENCY
CAUSALITY
HASSE DIAGRAM
STRATIFIED ORDER STRUCTURE
PARTIAL ORDER
INDEPENDENCE, PETRI NET
INHIBITOR ARC
COMPLEXITY
Hasse Diagrams of Combined Traces

Łukasz Mikulski$^1$ and Maciej Koutny$^2$

$^1$ Faculty of Mathematics and Computer Science
Nikolaus Copernicus University
Toruń, Chopina 12/18, Poland
frondo@mat.umk.pl

$^2$ School of Computing Science, Newcastle University
Newcastle upon Tyne, NE1 7RU, United Kingdom
maciej.koutny@ncl.ac.uk

Abstract. One of the standard ways to represent concurrent behaviours is to use concepts originating from language theory, such as traces and comtraces. Traces can express notions such as concurrency and causality, whereas comtraces can also capture weak causality and simultaneity. This paper is concerned with the development of efficient data structures and algorithms for manipulating comtraces. We introduce Hasse diagrams for comtraces which are a generalisation of Hasse diagrams defined for partial orders and traces, and develop an efficient algorithm for deriving them from language theoretic representations of comtraces. We also explain how the new representation of comtraces can be used to implement efficiently some basic operations on comtraces.

Keywords: comtrace, trace, concurrency, causality, Hasse diagram, stratified order structure, partial order, independence, Petri net, inhibitor arc, complexity

1 Introduction

The dynamic behaviours of concurrent systems, e.g., represented by Petri nets, are usually modelled as ongoing evolutions involving actions that take place at the interface with the environment. The simplest representations of such evolutions are sequences (or words) of executed actions, leading to a formal language semantics of Petri nets. However, words alone cannot express concurrency and causality between executed actions which are features of paramount importance if one wants to understand or efficiently analyse concurrent behaviours. To address this issue, one may consider adding an additional information about the relevant properties of behaviours, for example, in the form of causal dependencies between actions. This approach underpins the trace model of concurrent behaviour [1,9].

Consider, for example, the elementary net system [13] in Example 1(a). It is rather clear that two of its transitions, $a$ and $b$, are independent (or concurrent) and both cause the execution of the third transition, $c$. Therefore, the words $abc$ and $bac$ are essentially the same views of a concurrent behaviour in which one first independently executes $a$ and $b$, and then their execution causes $c$. This
interpretation is captured by the trace $[abc] = \{abc, bac\}$ which is an equivalence class of the relation capturing the intrinsic similarity of sequential executions of concurrent systems represented by elementary net systems. Moreover, such a trace can be seen as a (causal) partial order with three elements — one for each executed transition — in which $a$ and $b$ are unordered and both precede $c$.

Traces are not sufficient, however, when one needs to deal with elementary net systems with inhibitor arcs, such as that in Example 1(b). In this case, we allow sets of transitions (or steps) to be executed simultaneously. As a result, the net can execute step sequences $\{a\}\{b\}\{c\}$ and $\{a, b\}\{c\}$, but not $\{b\}\{a\}\{c\}$. Another example is shown in Example 1(c) where $\{a, b\}\{c\}$ is a possible execution, but neither $\{a\}\{b\}\{c\}$ nor $\{b\}\{a\}\{c\}$ is. To deal with elementary net systems with inhibitor arcs one may extend trace with additional information about intrinsic relationships between executed actions in the form of weak causality (where $a$ weakly precedes $b$ if it can be executed earlier or simultaneously with $b$). Hence, the situation we described for Example 1(b, c) can be captured by two equivalence classes of step sequences, called comtraces, $[\{a\}\{b\}\{c\}] = \{\{a\}\{b\}\{c\}, \{a, b\}\{c\}\}$ and $[\{a, b\}\{c\}] = \{\{a, b\}\{c\}\}$. The resulting model of comtraces [5, 7, 6] enjoys properties similar to that developed for traces. In particular, comtraces can be represented by so-structures which add weak causality to the standard causal partial orders.

Example 1. Three elementary net systems (with inhibitor arcs in case of (b) and (c)) of similar shape.

For system (a) traces are sufficient to capture concurrent behaviour, but for (b) and (c) we need comtraces.

In this paper, we are concerned with the development of efficient data structures and algorithms for manipulating comtraces. We introduce Hasse diagrams for comtraces which are a generalisation of Hasse diagrams [15] defined for partial orders and traces, and develop an efficient algorithm for deriving them from single step sequence representatives of comtraces. We also explain how the proposed representation of comtraces can be used to implement efficiently some basic operations on comtraces.
The paper is organised as follows. In Section 2, we provide basic notation and terminology. Section 3 recalls a number of notions and notations concerning traces, and sketches an algorithm for deriving Hasse diagram for the partial order induced by a trace. In Section 4 we discuss contraces and their graph representations (so-structures). This is followed up in Section 5 by an efficient construction of Hasse diagrams from step sequence representatives of contraces. Section 6 describes some applications of the newly proposed Hasse diagram of an so-structure. Finally, in Section 7, we summarise the contribution of this paper and briefly discuss directions for further research. Proofs of all the results can be found in [11].

2 Preliminaries

Throughout the paper we use the standard notions of the formal language theory. In particular, by an alphabet we mean a nonempty finite set Σ, the elements of which are called (atomic) actions. Finite sequences over Σ are called words. The sets of all finite words, including the empty word λ, is denoted by Σ∗.

Let \( w = a_1 \ldots a_n \) and \( v = b_1 \ldots b_m \) be two words. Then

\[
w \circ v = wz = a_1 \ldots a_nb_1 \ldots b_m
\]

is the concatenation of \( w \) and \( v \). The alphabet \( \text{alph}(w) \) of \( w \) is the set of all the actions occurring within \( w \), and \( \#_a(w) \) is the number of occurrences of an action \( a \) within \( w \). The set \( \text{occ}(w) \) of action occurrences of \( w \) comprises all pairs \( (a, i) \) such that \( a \in \text{alph}(w) \) and \( 1 \leq i \leq \#_a(w) \). The head (first action occurrences) and tail (last action occurrences) of a word \( w \) are two sets defined by:

\[
\text{head}(w) = \{(a, 1) \mid a \in \text{alph}(w)\}
\]
\[
\text{tail}(w) = \{(a, \#_a(w)) \mid a \in \text{alph}(w)\}.
\]

Let \( \alpha = (a, i) \) be an action occurrence in \( \text{occ}(w) \). The position \( \text{pos}_w(\alpha) \) of \( \alpha \) within \( w \) is the smallest integer \( j \) such that \( \#_a(a_1 \ldots a_j) = i \), and \( \ell(\alpha) = a \) is the default label of \( \alpha \). We can apply \( \ell \) to sequences and sets of action occurrences in the usual way, i.e.,

\[
\ell(\alpha_1 \ldots \alpha_n) = \ell(\alpha_1) \ldots \ell(\alpha_n) \quad \text{and} \quad \ell(\{\alpha_1, \ldots, \alpha_n\}) = \{\ell(\alpha_1), \ldots, \ell(\alpha_n)\}.
\]

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

\[
\prec^{\text{cov}} = R \setminus \bigcup_{i \geq 2} R^i.
\]
A linearisation of a poset $po = (X, \prec)$ is any sequence $u = x_1 \ldots x_n$ of distinct elements of $po$ such that $X = \{x_1, \ldots, x_n\}$ and, for all $1 \leq i < j \leq n$, $x_j \neq x_i$.

3 Traces

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

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

Equivalence classes of $\equiv_\Psi$ are called (Mazurkiewicz) traces (see [4,9,10]), and the trace containing a given word $w$ is denoted by $[w]$. The set of all traces over $\Psi$ is denoted by $\Sigma^*/\equiv_\Psi$, and the pair $(\Sigma^*/\equiv_\Psi, \circ)$ is a (trace) monoid, where $\tau \circ \tau' = [w \circ w']$, for any words $w \in \tau$ and $w' \in \tau'$, is the concatenation operation for traces. Note that trace concatenation is well-defined as $[w \circ w'] = [v \circ v']$, for all $w, v \in \tau$ and $w', v' \in \tau'$. Similarly, for every trace $\tau = [w]$ and every action $a \in \Sigma$, we can define:

$$\begin{align*}
\text{alph}(\tau) &= \text{alph}(w) & \text{occ}(\tau) &= \text{occ}(w) & \#_a(\tau) &= \#_a(w) \\
\text{head}(\tau) &= \text{head}(w) & \text{tail}(\tau) &= \text{tail}(w).
\end{align*}$$

Example 2. Consider a concurrent alphabet $\Psi$ with four actions $\Sigma = \{a, b, c, d\}$ together with a dependence relation $\text{dep}$ given by:

$$\begin{align*}
\text{dep} & \quad \text{or, equivalently,} \\
\begin{array}{c}
a \\
d
\end{array} & \quad \text{an independence relation} \\
\begin{array}{c}
b \\
c
\end{array}
\end{align*}$$

Note that in the example diagrams we will use $a_i$ to denote an action occurrence $(a, i)$, etc.

Then $w = abbaacd \equiv_\Psi abcaaad$. The set of action occurrences of $w$ is:

$$\text{occ}(w) = \{(a, 1), (a, 2), (a, 3), (b, 1), (b, 2), (c, 1), (d, 1)\} = \{a_1, a_2, a_3, b_1, b_2, c_1, d_1\}.$$  

Its head is $\{a_1, b_1, c_1, d_1\}$ and tail $\{a_3, b_2, c_1, d_1\}$.  

A word $w \in \Sigma^*$ is in (Foota) canonical form (see [3]) w.r.t. the dependence relation $\text{dep}$ and a lexicographic order $\text{lex}$ on $\Sigma$, if $w = w_1 \ldots w_n$ ($n \geq 0$), where each $w_j$ is a nonempty word such that:
− $w_i$ is fully commutative and minimal w.r.t. \textit{lex} among all sequences $u \in \Sigma^*$ satisfying $\text{occ}(w_i) = \text{occ}(u)$; and
− for each $i > 2$ and $a$ occurring in $w_i$, there is $b$ occurring in $w_{i-1}$ such that $(a, b) \in \text{dep}$.

Each trace contains exactly one sequence in canonical form, called its \textit{canonical representation}.

One can represent a trace $\tau$ as a poset of action occurrences. More precisely, $\text{po}(\tau) = (\text{occ}(w), \prec^w)$ is the poset induced by $\tau$, where $w$ is any word belonging to $\tau$, and $\prec_w$ is a binary relation on $\text{occ}(w)$ such that $\alpha \prec_w \beta$ if $\text{pos}_w(\alpha) < \text{pos}_w(\beta)$ and $(\ell(\alpha), \ell(\beta)) \in \text{dep}$. The soundness of this definition stems from the following

− $\tau = \{ \ell(u) \mid u \text{ is a linearisation of } \text{po}(\tau) \}$; and
− $(\text{occ}(w), \prec^w) = (\text{occ}(v), \prec^v)$, for all $w, v \in \tau$.

Note that $\prec_w = \prec_v$, for all words $w, v \in \tau$.

The \textit{Hasse diagram} of $\tau$, denoted by $H(\tau)$ (or $H(w)$, for any $w \in \tau$), is then simply the Hasse diagram of $\text{po}(\tau)$. Note that each arc of $H(w)$ belongs to $\prec_w$, but not necessarily vice versa.

### 3.1 Direct construction of Hasse diagrams for traces

We now sketch the idea behind an efficient way of generating Hasse diagrams for traces. The procedure takes an arbitrary word belonging to a trace together with the dependence relation.

The on-line algorithm of [12] scans the input word from left to right, and is based on an observation that when adding a new vertex to an already constructed Hasse diagram $H(v) = (\text{occ}(v), \prec^v)$ in order to generate $H(va) = (\text{occ}(va), \prec^{va})$, it suffices to consider only tail$(v)$, i.e., the latest occurrences of actions in $v$.

To use the relevant relationships involving such occurrences, for every action $a$ appearing in the input word, the algorithm maintains:

− $\text{ORD}_a$ which is the list of all actions dependent with $a$ which have been seen so far, in the reversed order of appearance of their latest occurrences; and
− $\text{VS}_a$ which is the set of all actions (called sources) whose latest occurrences are connected to the latest occurrence of $a$ (if the latter exists), i.e., $b \in \text{VS}_a$ if there is a path from $(b, \#_b(v))$ to $(a, \#_a(v))$ in $H(v)$.

When constructing $H(va)$ from an already built $H(v)$, the algorithm adds new arcs from tail$(v)$ to $(a, \#_a(v) + 1)$, and dynamically computes a new set of sources $\text{VS}^{\text{new}}_a$ of $a$ (initially, $\text{VS} = \emptyset$). The algorithm scans the list $\text{ORD}_a$ and, for each $b$ which is not a source of any of the sources of $a$ (i.e., $b \notin \bigcup_{c \in \text{VS}} \text{VS}_c$), generates a new arc from $(b, \#_b(v))$ to $(a, \#_a(v) + 1)$ and sets $\text{VS}$ to $\text{VS} \cup \text{VS}_b$. Moreover, after each step the data structure that describes the tail is updated; in particular, each $\text{VS}_b$ with $b \neq a$ is set to $\text{VS}_b \setminus \{a\}$ (see Example 3). The correctness of whole procedure is guaranteed by respecting the generation order of the latest occurrences of actions stored in $\text{ORD}_a$. 
The algorithm has time complexity of $O(nk^2)$, where $n$ is the number of action occurrences in the input word, and $k$ is the number of different actions appearing in the input word. Typically, $k$ (bounded by $|\Sigma|$) is much smaller than $n$ and in any case it is fixed for a given concurrent system (e.g., a Petri net). For practical applications the above algorithm can therefore be considered as linear in the size of its input, and so optimal as any algorithm generating the Hasse diagram of a trace must look at least once at each of its action occurrences.

Example 3. Consider two words, $v = bbacad$ and $vb = bbacadb$, over the concurrent alphabet from Example 2. Below we show the diagrams of the corresponding Hasse diagrams, indicating in bold the action occurrences belonging to $tail(v)$ and $tail(vb)$, respectively, together with auxiliary tail data structures.

\[
\begin{aligned}
 & b_1 \rightarrow b_2 \rightarrow a_1 \rightarrow c_1 \rightarrow a_2 \rightarrow d_1 \\
 & \text{ORD} \quad \text{VS} \\
 a & \quad (d, a, b) \quad \{a, b\} \\
 b & \quad (a, c, b) \quad \{b\} \\
 c & \quad (d, c, b) \quad \{c, b\} \\
 d & \quad (d, a, c) \quad \{a, b, c, d\}
\end{aligned}
\]

\[
\begin{aligned}
 & b_1 \rightarrow b_2 \rightarrow a_1 \rightarrow c_1 \rightarrow a_2 \rightarrow d_1 \rightarrow b_3 \\
 & \text{ORD} \quad \text{VS} \\
 a & \quad (b, d, a) \quad \{a\} \\
 b & \quad (b, a, c) \quad \{a, b, c\} \\
 c & \quad (b, d, c) \quad \{c\} \\
 d & \quad (d, a, c) \quad \{a, c, d\}
\end{aligned}
\]

\[\square\]

4 Comtraces

A comtrace alphabet is a triple $\Theta = (\Sigma, \sim, \ser)$, where $\Sigma$ is an alphabet and $\ser \subseteq \sim \subseteq \Sigma \times \Sigma$ are two relations, respectively called serialisability and simultaneity; it is assumed that $\sim$ is irreflexive and symmetric. Intuitively, if $(a, b) \in \sim$ then $a$ and $b$ may occur simultaneously, whereas $(a, b) \in \ser$ means that in such a case $a$ may also occur before $b$ (with both executions being equivalent). The set of all (potential) steps over $\Theta$, or step alphabet, is then defined as the set $S$ comprising all nonempty sets of actions $A \subseteq \Sigma$ such that $(a, b) \in \sim$, for all distinct $a, b \in A$. Finite sequences in $S^*$, including the empty one denoted by $\lambda$, are called step sequences.
We will now lift a number of notions and notations introduced for words to the level of step sequences. In what follows, \( \Theta = (\Sigma, \text{sim}, \text{ser}) \) is a fixed contrace alphabet.

Let \( w = A_1 \ldots A_n \) and \( v = B_1 \ldots B_m \) be two step sequences. Then \( w \circ v = wv = A_1 \ldots A_n B_1 \ldots B_m \) is the concatenation of \( w \) and \( v \). The alphabet \( \alpha(w) \) of \( w \) comprises all action occurring within \( w \), and \( \#_a(w) \) is the number of occurrences of an action \( a \) within \( w \). The set \( \text{occ}(w) \) of action occurrences of \( w \) comprises all pairs \( (a,i) \) with \( a \in \alpha(w) \) and \( 1 \leq i \leq \#_a(w) \). The position \( \text{pos}_a(\alpha) \) of an action occurrence \( \alpha = (a,i) \in \text{occ}(w) \) is given as the smallest integer \( j \) such that \( \#_a(A_1 \ldots A_j) = i \). In such a case, we also denote \( \alpha \in \text{occ}_j(w) \). Hence the sets \( \text{occ}_1(w), \ldots, \text{occ}_n(w) \) form a partition of \( \text{occ}(w) \). We can also apply the default labelling \( \ell \) to (sequences of) sets of action occurrences in the usual way. The head and tail of \( w \) are given by:

\[
\text{head}(w) = \{(a,1) \mid a \in \alpha(w)\} \\
\text{tail}(w) = \{(a,\#_a(w)) \mid a \in \alpha(w)\}.
\]

The \emph{contrace congruence} over \( \Theta \), denoted by \( \equiv_\Theta \), is the reflexive, symmetric and transitive closure of the relation \( \sim_\Theta \subseteq S^* \times S^* \) such that \( w \sim_\Theta v \) if there are \( u,z \in S^* \) and \( A,B,C \in S \) satisfying \( w = uAz \), \( v = uBCz \), \( A = B \cup C \) and \( B \times C \subseteq \text{ser} \). Note that \( B \cap C = \emptyset \) as \( \text{ser} \) is irreflexive.

Equivalence classes of the relation \( \equiv_\Theta \) are called \emph{contraces} (see [6]), and the contrace containing a given step sequence \( w \) is denoted by \( [w] \). The set of all contraces is denoted by \( S^*/\equiv \), and the pair \((S^*/\equiv, \circ)\) is a (contrace) monoid, where \( \tau \circ \tau' = [w \circ w'] \), for any step sequences \( w \in \tau \) and \( w' \in \tau' \). Contrace concatenation is well-defined as \( [w \circ w'] = [v \circ v'] \), for all \( w,v \in \tau \) and \( w',v' \in \tau' \).

A contrace \( \tau \) is a prefix of a contrace \( \tau' \) if there is a contrace \( \tau'' \) such that \( \tau \circ \tau'' = \tau' \). As in the case of traces, for every contrace \( \tau \) and every \( a \in \Sigma \), we can define \( \alpha(\tau) = \alpha(w), \#_a(\tau) = \#_a(w), \text{occ}(\tau) = \text{occ}(w), \text{head}(\tau) = \text{head}(w) \) and \( \text{tail}(\tau) = \text{tail}(w) \), where \( w \) is any step sequence belonging to \( \tau \).

Next, we give the canonical form of a contrace which essentially captures a greedy, maximally concurrent, execution of the actions occurring in the contrace conforming to the simultaneity and serialisability relations. A step sequence \( w = A_1 \ldots A_n \in S^* \) is in (Foata) \emph{canonical form} if, for each \( i \leq n \), whenever \( Av \equiv_\Theta A_1 \ldots A_k \) for some \( A \in S \) and \( v \in S^* \), then \( |A| \leq |A_i| \). One can see that each contrace comprises a unique step sequence in normal form. Note that an alternative (equivalent) definition of normal form requires that, for every \( i < k \), there is no \( A \neq A \subseteq A_{i+1} \) such that \( A_1 \times A \subseteq \text{ser} \) and \( A \times (A_{i+1} \setminus A) \subseteq \text{ser} \).

In technical discussion, we will use four relations covering all possible relationships between individual actions:

- Dependence \( \text{dep} = (\Sigma \times \Sigma) \setminus \text{sim} \), and independence \( \text{ind} = \text{ser} \cap \text{ser}^{-1} \).

Both relations have their counterparts in trace theory, and so we denote them in the same way. If two actions are dependent then their two occurrences must happen in the same order (and never simultaneously) in all the step sequences forming a given contrace. Two actions are independent if they can be executed in any order as well as simultaneously (as \( \text{ser} \subseteq \text{sim} ) \).
- Strong simultaneity $ssm = sim \setminus (ser \cup ser^{-1})$.
  If two actions are strongly simultaneous then their two occurrences must
  happen in the same order or simultaneously in all the step sequences forming
  a given contrace.
- Weak dependence $wdp = ser \setminus ser^{-1}$.
  Two actions are weakly dependent if they can be serialised only in one way;
  hence this relationship is antisymmetric.

Example 4. Consider a contrace alphabet $\Theta$ with four actions $\Sigma = \{a, b, c, d\}$
  together with a simultaneity and serialisability relations, $ser$ and $sim$ given by:

$\begin{align*}
  sim &= a \longrightarrow b \\
  ser &= a \rightsquigarrow b \\
  ind &= a \longrightarrow b \\
  dep &= a \rightsquigarrow b
\end{align*}$

Then the four derived relations on actions are as follows:

$\begin{align*}
  ind &= a \longrightarrow b \\
  dep &= a \rightsquigarrow b \\
  ssm &= a \longrightarrow b \\
  wdp &= a \longrightarrow b
\end{align*}$

Then $[w] = [u]$, where:

$\begin{align*}
  w &= \{d\} \{a, b\} \{c\} \{d\} \{a, b, c\} \{c\} \\
  u &= \{a, d\} \{b, c\} \{d\} \{a, b, c\} \{c\}
\end{align*}$

Moreover, $u$ is a step sequence in normal form. $\square$

4.1 Stratified order structures

Contraces can be represented by so-structures, in a similar way as traces can
be represented by posets.

A stratified order structure (or so-structure) is a tuple $sos = (X, \prec, \sqsubset)$ com-
prising two binary relations, $\prec$ (causality) and $\sqsubset$ (weak causality), on a finite
set $X$ such that, for all $x, y, z \in X$:

$\begin{align*}
  S1 : & \quad x \not\sqsubset x \\
  S2 : & \quad x \prec y \implies x \sqsubset y \\
  S3 : & \quad x \sqsubset y \sqsubset z \land x \neq z \implies x \sqsubset z \\
  S4 : & \quad x \sqsubset y \lor y \prec x \sqsubset z \implies x \prec z.
\end{align*}$

Intuitively, $\prec$ represents the ‘earlier than’ relationship in $X$, and $\sqsubset$ the ‘not later
than’ relationship. Note that $\prec$ is a partial order, and $x \prec y$ implies $y \not\sqsubset x$. In a
diagrammatical representation, $\sqsubset$ is represented by dashed arcs.
The so-closure of a triple \( q = (X, \prec, \sqsubseteq) \), where \( X \) is a finite set and \( \prec, \sqsubseteq \) are binary relations on \( X \) is defined as \( q^\circ = (X, \gamma \circ \prec \circ \gamma, \gamma \setminus \text{id}_X) \), where \( \gamma = (\prec \cup \sqsubseteq)^* \). If \( q^\circ = \text{sos} \), where \( \text{sos} \) is an so-structure, then \( \rho \) is called an sos-diagram.

A stratification of an so-structure \( \text{sos} = (X, \prec, \sqsubseteq) \) is any sequence \( u = X_1 \ldots X_n \) of nonempty disjoint subsets of \( X \) such that \( X = X_1 \cup \ldots \cup X_n \),

- \( (X_j \times X_i) \cap \prec = \emptyset \), for all \( 1 \leq i \leq j \leq n \); and
- \( (X_j \times X_i) \cap \sqsubseteq = \emptyset \), for all \( 1 \leq i < j \leq n \).

The so-structure induced by a contrace \( \tau \) is defined as \( \text{sos}(\tau) = (\text{occ}(w), \prec_w, \sqsubseteq_w)^\circ \),

where \( w \) is any step sequence \( w \in \tau \), and:

\[
\prec_w = \{(\alpha, \beta) \in \text{occ}(w) \times \text{occ}(w) \mid \text{pos}_w(\alpha) < \text{pos}_w(\beta) \land (\ell(\alpha), \ell(\beta)) \not\in \text{ser}\} \\
\sqsubseteq_w = \{(\alpha, \beta) \in \text{occ}(w) \times \text{occ}(w) \mid \text{pos}_w(\alpha) \leq \text{pos}_w(\beta) \land (\ell(\beta), \ell(\alpha)) \not\in \text{ser}\}.
\]

Note that \( \prec_w = \prec_v \) and \( \sqsubseteq_w = \sqsubseteq_v \), for all \( w, v \in \tau \).

The soundness of the last definition stems from the following:

- \( \tau = \{\ell(v) \mid v \text{ is a stratification of } \text{sos}(\tau)\} \), and
- \( (\text{occ}(w), \prec_w, \sqsubseteq_w)^\circ = (\text{occ}(v), \prec_v, \sqsubseteq_v)^\circ \), for all step sequences \( w, v \in \tau \).

### 4.2 Folded so-structures

Weak causality is a pre-order rather than a partial order relation, and it can be advantageous to work with a quotient so-structure derived from \( \text{sos}(\tau) = (\text{occ}(\tau), \prec, \sqsubseteq) \) induced by a contrace \( \tau \). First, for each action occurrence \( \alpha \in \text{occ}(\tau) \), we denote by \( \langle \alpha \rangle \) the equivalence class of the \( \sqsubseteq \)-cycle relation comprising \( \alpha \), i.e., \( \alpha \) together with the set of all \( \beta \in \text{occ}(\tau) \) satisfying \( \alpha \sqsubseteq \beta \sqsubseteq \alpha \). Each such \( \Delta = \langle \alpha \rangle \) will be called a folded action and their set denoted by \( \widehat{\text{occ}}(\tau) \). Then the folded so-structure induced by a contrace \( \tau \) is defined as \( \widehat{\text{sos}}(\tau) = (\widehat{\text{occ}}(\tau), \hat{\prec}, \hat{\sqsubseteq}) \), where, for all \( \Delta, \Delta' \in \widehat{\text{occ}}(\tau) :

- \( \Delta \hat{\preceq} \Delta' \) if \( (\Delta \times \Delta') \cap \prec \neq \emptyset \); and
- \( \Delta \hat{\sqsubseteq} \Delta' \) if \( (\Delta \times \Delta') \cap \sqsubseteq \neq \emptyset \) and \( \Delta \neq \Delta' \).

Note that, by \( S_4 \) and \( \Delta \hat{\preceq} \Delta', \Delta \times \Delta' \) is included in \( \prec \), and, by \( S_3 \) and \( \hat{\Delta} \hat{\sqsubseteq} \Delta' \), \( \Delta \times \Delta' \) is included in \( \sqsubseteq \).

By \( S_2-S_4 \), \( \widehat{\text{sos}}(\tau) \) is an so-structure, and \( \hat{\sqsubseteq} \) is a poset containing \( \hat{\prec} \). Moreover, different contraces induce different folded so-structures, and there is a straightforward way of recovering \( \text{sos}(\tau) \) from \( \widehat{\text{sos}}(\tau) \) as we have:

- \( \alpha \prec \beta \) iff \( \langle \alpha \rangle \hat{\prec} \langle \beta \rangle \), and
- \( \alpha \sqsubseteq \beta \) iff \( \langle \alpha \rangle \hat{\sqsubseteq} \langle \beta \rangle \), or \( \alpha \neq \beta \land \langle \alpha \rangle = \langle \beta \rangle \).

It turns out that action occurrences occurring in a single step sequence can be partitioned into folded actions.
Proposition 5. Let \( w = A_1 \ldots A_n \) be a step sequence, \( i \leq n \), and SCC be the set of strongly connected components of the directed graph
\[
(\text{occ}_i(w), \subseteq_w |_{\text{occ}_i(w)}).
\]
Then SCC is a set of folded actions partitioning \( \text{occ}_i(w) \), and \( \sqsubseteq |_{\text{SCC} \times \text{SCC}} \) is an acyclic relation. Moreover, if we take any linearisation \((TS_1, \ldots, TS_m)\) of SCC then each comtrace \([A_1 \ldots A_{i-1}TS_1 \ldots TS_j]\), for \( j \leq m \), is a prefix of \([w]\).

Proof. We observe that, for all \( \alpha \in \text{occ}_i(w) \) and \( \beta \in \text{occ}(w) \), if \( \alpha \subseteq_w^+ \beta \subseteq_w^+ \alpha \) then \( \langle \alpha \rangle \subseteq \text{occ}_i(w) \). Hence each strongly connected component in SCC is a folded action. The second and third parts of the result follow directly from the definitions. \( \square \)

Folded so-structures can also be used to recover in a direct way all the step sequences belonging to a given comtrace.

Proposition 6. Let \( \tau \) be a comtrace.
Then we have that \( \tau = \{ \hat{\ell}(v) \mid v \text{ is a stratification of } s\hat{o}s(\tau) \} \), where:
\[
\hat{\ell}(v) = \left( \bigcup_{\Delta \in \Gamma_1} \ell(\Delta) \right) \ldots \left( \bigcup_{\Delta \in \Gamma_k} \ell(\Delta) \right),
\]
for every stratification \( v = \Gamma_1 \ldots \Gamma_k \) of \( s\hat{o}s(\tau) \).

Proof. \((\supseteq)\) Let \( v = \Gamma_1 \ldots \Gamma_k \) be a stratification of \( s\hat{o}s(\tau) \):
\[
w = A_1 \ldots A_k = \left( \bigcup_{\Delta \in \Gamma_1} \Delta \right) \ldots \left( \bigcup_{\Delta \in \Gamma_k} \Delta \right).
\]

We have that \( \Gamma_1, \ldots, \Gamma_k \) are non-empty disjoint sets of folded actions such that \( \text{occ}(\tau) = \Gamma_1 \cup \ldots \cup \Gamma_k \),
- \((\Gamma_i \times \Gamma_j) \cap \subseteq = \emptyset\), for all \( 1 \leq i \leq j \leq k \); and
- \((\Gamma_i \times \Gamma_j) \cap \subseteq = \emptyset\), for all \( 1 \leq i < j \leq k \).

Moreover, each folded action occurring in \( v \) is an equivalence class, and so different folded actions occurring in \( v \) are disjoint sets. Hence \( A_1, \ldots, A_k \) are non-empty disjoint sets of actions occurrences such that \( \text{occ}(\tau) = A_1 \cup \ldots \cup A_k \),
- \((A_i \times A_j) \cap \subseteq = \emptyset\), for all \( 1 \leq i \leq j \leq k \); and
- \((A_i \times A_j) \cap \subseteq = \emptyset\), for all \( 1 \leq i < j \leq k \).

As a result, \( w \) is a stratification of \( s\hat{o}s(\tau) \) satisfying \( \hat{\ell}(v) = \ell(w) \). Moreover, we have \( \tau = \{ \ell(w) \mid w \text{ is a stratification of } s\hat{o}s(\tau) \} \). Hence \( \hat{\ell}(v) \in \tau \).

\((\subseteq)\) To show the reverse inclusion, assume that \( w = A_1 \cup \ldots \cup A_k \) is a stratification of \( s\hat{o}s(\tau) \). Then each \( A_i \) can be partitioned into a set of folded actions \( \Gamma_i \) (see Proposition 5). One can then see that \( v = \Gamma_1 \ldots \Gamma_k \) is a stratification of \( s\hat{o}s(\tau) \) satisfying \( \hat{\ell}(v) = \ell(w) \). The inclusion then follows from \( \tau = \{ \ell(w) \mid w \text{ is a stratification of } s\hat{o}s(\tau) \} \). \( \square \)
The next result shows that folded actions can be derived directly from the step sequences forming a contrace.

**Proposition 7.** Let $\alpha$ and $\beta$ be two action occurrences of a contrace $\tau$. Then $\langle \alpha \rangle = \langle \beta \rangle$ iff $\text{pos}_w(\alpha) = \text{pos}_w(\beta)$, for every step sequence $w \in \tau$.

**Proof.** ($\Longrightarrow$) $\langle \alpha \rangle = \langle \beta \rangle$ implies that, for every stratification $v = X_1 \ldots X_n$ of $\text{sos}(\tau)$, $\alpha$ and $\beta$ belong to the same set $X_i$. Hence $\alpha$ and $\beta$ occur in the same set $\ell(X_i)$ in $\ell(v)$. Thus, by $\tau = \{ \ell(v) \mid v \text{ is a stratification of } \text{sos}(\tau) \}$, $\text{pos}_w(\alpha) = \text{pos}_w(\beta)$, for every step sequence $w \in \tau$.

($\Longleftarrow$) Let $\text{sos}(\tau) = (\text{occ}(\tau), \prec, \sqsubset)$. The implication follows from the fact that if $\alpha \npreceq \beta$ then there is a step sequence $w \in \tau$ such that $\text{pos}_w(\alpha) > \text{pos}_w(\beta)$.

### 4.3 Hasse diagrams of contraces

As a folded so-structure $\tilde{\text{sos}}(\tau) = (\tilde{\text{occ}}(\tau), \tilde{\prec}, \tilde{\sqsubset})$ comprises two nested posets, defining the *folded Hasse diagram* of a contrace $\tau$ is straightforward:

$$
\tilde{H}(\tau) = (\tilde{\text{occ}}(\tau), \tilde{\prec}, \tilde{\sqsubset}),
$$

where:

$$
\tilde{\prec} = \prec \setminus \{(\tilde{\circ} \circ \tilde{\prec}) \cup (\tilde{\circ} \circ \tilde{\sqsubset}) \cup (\tilde{\circ} \circ \prec)\},
$$

$$
\tilde{\sqsubset} = (\prec \setminus (\circ \circ \tilde{\sqsubset})) \setminus \tilde{\prec}.
$$

In particular, one can see that $\tilde{H}(\tau)$ is the smallest triple $(\tilde{\text{occ}}(\tau), \tilde{\prec}, \tilde{\sqsubset})$ whose so-closure yields $\tilde{\text{sos}}(\tau)$; in other words, $\tilde{H}(\tau)$ is the smallest $\tilde{\text{sos}}(\tau)$-diagram. Note that if $(\tilde{\text{occ}}(\tau), \tilde{\prec}, \tilde{\sqsubset})$ is an $\tilde{\text{sos}}(\tau)$-diagram and $\tilde{\sqsubset}$ is irreflexive, then $\tilde{\prec}$ and $\tilde{\sqsubset}$ are respectively included in $\prec$ and $\sqsubset \setminus \prec$.

If $\Delta$ and $\Delta'$ are two distinct vertices of an $\tilde{\text{sos}}(\tau)$-diagram $(\text{occ}(\tau), \prec, \sqsubset)$, then we say that:

- there is a *strong path* from $\Delta$ to $\Delta'$ if there are $\Delta = \Delta_1, \ldots, \Delta_n = \Delta'$ such that, for every $i < n$, $\Delta_i \tilde{\prec} \Delta_{i+1}$ or $\Delta_i \tilde{\sqsubset} \Delta_{i+1}$; moreover, $\Delta_j \tilde{\prec} \Delta_{j+1}$ for at least one $j < n$.
- there is a *weak path* from $\Delta$ to $\Delta'$ if there are $\Delta = \Delta_1, \ldots, \Delta_n = \Delta'$ such that, for every $i < n$, $\Delta_i \tilde{\prec} \Delta_{i+1}$; moreover, there is no strong path from $\Delta$ to $\Delta'$.

Note that if there is a weak path from $\Delta$ to $\Delta'$ then there is no weak path from $\Delta'$ to $\Delta$.

As far as the original so-structure induced by $\tau$ is concerned, the *Hasse diagram* of $\tau$ is defined as:

$$
H(\tau) = (\text{occ}(\tau), \prec, \sqsubset),
$$

where $\alpha \prec^h \beta$ if $\langle \alpha \rangle \prec^h \langle \beta \rangle$, and $\alpha \preceq^h \beta$ if $\langle \alpha \rangle \preceq^h \langle \beta \rangle$ or $\alpha \neq \beta$ and $\langle \alpha \rangle = \langle \beta \rangle$. Note that $H(\tau)$ is not guaranteed to be the smallest sos(τ)-diagram because,
in general, a minimal $sos(\tau)$-diagram does not exist as weak causality is only a pre-order.

Henceforth we will work with two different yet equivalent representations of contraces, as well as two different kinds of Hasse diagrams, always choosing that which is more convenient for technical considerations.

**Example 8.** The following is the $sos(w)$-diagram $(occ(w), \lessdot_w, \sqsubseteq_w)$ for the contrace $[w]$ from Example 4:

![Hasse diagram](image)

and its Hasse diagram is:

![Hasse diagram](image)

Note that $\{a_2, c_2\}$ indicated by a border in the above diagram is the only non-singleton folded action.

We now provide some general properties of Hasse diagrams of contraces. In what follows, for every step sequence $w$ in a contrace $\tau$, we denote $\hat{H}(w) = \hat{H}(\tau)$ and $H(w) = H(\tau)$.

**Proposition 9.** Let $\tau$ and $\tau'$ be two contraces. Then the (folded) Hasse diagrams of $\tau$ and $\tau'$ are vertex-induced subgraphs of the (folded) Hasse diagram of $\tau \circ \tau'$, assuming that in $(\hat{H}(\tau'))$ $H(\tau')$ each action occurrence $(a, i)$ has been replaced by $(a, i + \#_a(\tau'))$.

**Proof.** Follows from the definitions. We only observe that each folded action of $\hat{H}(\tau \circ \tau')$ is either a folded action in $\hat{H}(\tau)$, or a a folded action in $\hat{H}(\tau')$ with each action occurrence $(a, i)$ replaced by $(a, i + \#_a(\tau))$ (see Proposition 7).

**Corollary 10.** For every step sequence $wA$, the vertices of $\hat{H}(w)$ and $H(w)$ are also vertices of $\hat{H}(wA)$ and $H(wA)$, respectively.
Proposition 11. Let \( w \) be a step sequence, \((\langle a, i \rangle, \langle \beta \rangle)\) be an arc in \( \hat{H}(w) \), 
\((a, \ell(\beta)) \notin \text{ind} \), and \( \#_a(w) > i \). Then \( \text{pos}_w(a, i + 1) \geq \text{pos}_w(\beta) \).

Proof. Suppose that \( \text{pos}_w(a, i + 1) < \text{pos}_w(\beta) \) which implies \( \text{pos}_w(a, i) < \text{pos}_w(\beta) \). Then, by Proposition 7, \((\langle a, i \rangle, \langle a, i + 1 \rangle) \neq \langle \beta \rangle \). We also have \((a, i + 1) \prec_w \beta \) or \((a, i + 1) \sqsubset_w \beta \), and so there is a path from \( \langle (a, i) \rangle \) to \( \langle \beta \rangle \) in \( \hat{H}(w) \). We then observe that \((a, i) \prec_w (a, i + 1) \) and \((\langle a, i \rangle) \neq \langle (a, i + 1) \rangle \), and so there is a strong path from \( \langle (a, i) \rangle \) to \( \langle (a, i + 1) \rangle \) in \( \hat{H}(w) \). Hence there is no arc \((\langle (a, i), \langle \beta \rangle \rangle) \) in \( \hat{H}(w) \), yielding a contradiction. \( \square \)

Proposition 12. Let \( w \) be a step sequence. Moreover, let \( \Delta \) and \( \Delta' \) be two distinct vertices of \( \hat{H}(w) = (\hat{\alpha}c\hat{c}, \hat{x}^h, \hat{x}^h) \).

1. If \( (\Delta, \Delta') \in \hat{x}^h \), then there are \( \alpha \in \Delta \) and \( \alpha' \in \Delta' \) such that \( \alpha \prec_w \alpha' \).
2. If \( (\Delta, \Delta') \in \hat{x}^h \), then there are \( \alpha \in \Delta \) and \( \alpha' \in \Delta' \) such that \( \alpha \subset_w \alpha' \), and there are no \( \beta \in \Delta \) and \( \beta' \in \Delta' \) such that \( \beta \prec_w \beta' \).
3. If there are \( \alpha \in \Delta \) and \( \alpha' \in \Delta' \) such that \( \alpha \prec_w \alpha' \), then the following statements are equivalent:
   - \((\Delta, \Delta') \notin \hat{x}^h \).
   - In \( \hat{H}(w) \), there is a vertex \( \Delta'' \) different from \( \Delta \) and \( \Delta' \) such that there is a weak or strong path from \( \Delta \) to \( \Delta'' \), and a weak or strong path from \( \Delta'' \) to \( \Delta' \); moreover, at least one of these paths is strong.
4. If there are \( \alpha \in \Delta \) and \( \alpha' \in \Delta' \) such that \( \alpha \subset_w \alpha' \) and there are no \( \beta \in \Delta \) and \( \beta' \in \Delta' \) such that \( \beta \prec_w \beta' \), then the following statements are equivalent:
   - \((\Delta, \Delta') \notin \hat{x}^h \).
   - In \( \hat{H}(w) \), there is a vertex \( \Delta'' \) different from \( \Delta \) and \( \Delta' \) such that there is a weak or strong path from \( \Delta \) to \( \Delta'' \), and a weak or strong path from \( \Delta'' \) to \( \Delta' \).

Proof. (1) and (2) follow directly from the definitions. We only observe that if \( \alpha \in \Delta \) and \( \alpha' \in \Delta' \) are such that \( \alpha \prec_w \alpha' \) then \( (\Delta, \Delta') \in \hat{\alpha}x \), and if \( \alpha \in \Delta \) and \( \alpha' \in \Delta' \) are such that \( \alpha \subset_w \alpha' \) then \( (\Delta, \Delta') \in \hat{\alpha}x \).

(3) Suppose that \( \alpha \in \Delta \), \( \alpha' \in \Delta' \) and \( \alpha \prec_w \alpha' \). If \( (\Delta, \Delta') \notin \hat{x}^h \) then, directly from the definition,
\[
(\Delta, \Delta') \in (\hat{\alpha} \circ \hat{x}) \cup (\hat{\rho} \circ \hat{x}) \cup (\hat{\mu} \circ \hat{\alpha}) ,
\]
and so there exists \( \Delta'' \) such that \( \Delta \hat{\rho} \Delta'' \hat{\rho} \Delta' \) or \( \Delta \hat{\rho} \Delta'' \hat{\rho} \Delta' \) or \( \Delta \hat{\mu} \Delta'' \hat{\mu} \Delta' \). As \( \hat{\alpha}x \) is a subset of \( \hat{x} \), we have \( \Delta \hat{\mu} \Delta'' \hat{\mu} \Delta' \) and \( \Delta \hat{\rho} \Delta'' \hat{\rho} \Delta' \), so there is a weak or strong path from \( \Delta \) to \( \Delta'' \), and a weak or strong path from \( \Delta'' \) to \( \Delta' \), and at least one of these paths is strong.

To show the reverse implication, suppose that there is a vertex \( \Delta'' \) different from \( \Delta \) and \( \Delta' \) such that there is a weak or strong path from \( \Delta \) to \( \Delta'' \), and a weak or strong path from \( \Delta'' \) to \( \Delta' \), and at least one of these paths is strong. Then \( (\Delta, \Delta') \in (\hat{\rho} \circ \hat{x}) \cup (\hat{\rho} \circ \hat{x}) \cup (\hat{\mu} \circ \hat{\alpha}) \), and so \( (\Delta, \Delta') \notin \hat{x}^h \).

(4) Similar to the proof of (3). \( \square \)
Corollary 13. For every step sequence \( wA, \hat{H}(w) \) and \( H(w) \) are vertex-induced subgraphs of \( \hat{H}(wA) \) and \( H(wA) \), respectively.

Proposition 14. Let \( wA \) be a step sequence and:
\[
\hat{H}(w) = (\hat{o}\sigma\sigma(w), \hat{z}^h_w, \hat{e}^h_w) \quad \text{and} \quad \hat{H}(wA) = (\hat{o}\sigma\sigma(wA), \hat{z}^h_{wA}, \hat{e}^h_{wA})
\]
be the folded Hasse diagrams of \( w \) and \( wA \), respectively. Then:
\[
- \hat{z}^h_{wA} \setminus \hat{z}^h_w \subseteq \hat{\text{tail}}(w) \times (\hat{o}\sigma\sigma(wA) \setminus \hat{o}\sigma\sigma(w)), \quad \text{and}
- \hat{e}^h_{wA} \setminus \hat{e}^h_w \subseteq (\hat{\text{tail}}(w) \cup (\hat{o}\sigma\sigma(wA) \setminus \hat{o}\sigma\sigma(w))) \times (\hat{o}\sigma\sigma(wA) \setminus \hat{o}\sigma\sigma(w)),
\]
where \( \hat{\text{tail}}(w) = \{ \Delta \in \hat{o}\sigma\sigma(w) \mid \Delta \cap \hat{\text{tail}}(w) \neq \emptyset \} \).

Proof. Follows directly from the definitions. We only observe that if \( (\Delta, \Delta') \in \hat{z}^h_{wA} \) then, by Proposition 12, there are \( \alpha \in \Delta \) and \( \alpha' \in \Delta' \) such that \( \alpha \prec_w \alpha' \).

Moreover, if \( (\Delta, \Delta') \notin \hat{z}^h_w \) then \( \alpha' \in \text{occ}_{wA}(wA) \). Hence, if \( \alpha \neq (\ell(\alpha), \#(\ell(\alpha))(wA)) \) then we obtain a contradiction with Proposition 11.

The case \( (\Delta, \Delta') \in \hat{z}^h_{wA} \setminus \hat{z}^h_w \) is similar. \( \square \)

4.4 Constructing contrace graphs

Given a step sequence \( w \) with \( |\text{occ}(w)| = n \), perhaps the easiest way of constructing the graph of the Hasse diagram \( H(w) = H(\tau) = (\text{occ}(w), \prec, \sqsubset) \) induced by the contrace \( \tau = [w] \) is to simply follow the definitions. First, we take the set of action occurrences \( \text{occ}(w) \) to be the vertices of \( H(w) \), and process one-by-one all possible pairs of distinct vertices. For each such pair \( (\alpha, \beta) \), we check whether \( \text{pos}_w(\alpha) \prec \text{pos}_w(\beta) \) and \( (\ell(\alpha), \ell(\beta)) \notin \text{ser} \) (which yields \( \alpha \prec_w \beta \)), or whether \( \text{pos}_w(\alpha) \leq \text{pos}_w(\beta) \) and \( (\ell(\beta), \ell(\alpha)) \notin \text{ser} \) (which yields \( \alpha \sqsubset_w \beta \)). This can be done in \( O(n^2) \) time, and the resulting graph \( (\text{occ}(w), \prec_w, \sqsubset_w) \) has the size \( O(n^2) \). Example 4 shows the result of carrying such checks for the contrace \( [w] \) of Example 4.

Next, we apply the so-closure to get \( \text{soc}(w) = (\text{occ}(w), \prec, \sqsubset) \), and then remove all the arcs implied by \( S2 \rightarrow S4 \), in order to generate \( H(w) \). Applying the so-closure operation is a straightforward generalisation of the transitive closure of a binary relation. Removing unnecessary arcs, however, is not so. First, we temporarily remove all the arcs joining the action occurrences belonging to the same folded action, obtaining \( \sqsubset' = \sqsubset \setminus (\sqsubset \cap \sqsubset^{-1}) \). Then, we delete from \( \prec \) all the arcs \( (\alpha, \beta) \) for which there exists \( \delta \) satisfying \( \alpha \prec \delta \prec \beta \) or \( \alpha \prec \delta \sqsubset \beta \), obtaining \( \prec^h \). Then we delete from \( \sqsubset' \) all the arcs \( (\alpha, \beta) \) belonging to \( \prec^h \), and those for which there exists \( \delta \) satisfying \( \alpha \sqsubset \delta \sqsubset \beta \), obtaining \( \sqsubset'' \). Finally, \( \sqsubset^h \) is calculated as \( \sqsubset'' \cup (\sqsubset \cap \sqsubset^{-1}) \).

As far as complexity is concerned, as we computed \( (\text{occ}(w), \prec_w, \sqsubset_w) \), the overall complexity of the algorithm is at least linear in terms of \( n \). In the rest of the paper, we will provide much better solution that takes advantage of the properties of the Hasse diagrams of consecutive prefixes of a given step sequence, resulting in what can be regarded as a linear algorithm.
5 Direct Construction of Hasse Diagrams of Comtraces

In this section we will show how to construct the (folded) Hasse diagram directly from a given representative \( v \) of a comtrace. More precisely, the input to the algorithm is a comtrace alphabet \( (\Sigma, \text{sim}, \text{ser}) \) and a step sequence \( v \in \Sigma^* \). We first describe the algorithm and provide its pseudo-code. After that, we discuss its complexity.

The algorithm is on-line (i.e., its successive phases generate correct Hasse diagrams for all the prefixes of \( v \)), and exploits the knowledge of the structure of the intermediate diagrams. A key observation concerns the tail of an intermediate folded Hasse diagram (denoted as \( \text{tail} \)) captured by Proposition 14. Such a tail comprises the latest occurrences of each action which has so far been ‘seen’ by the algorithm. Only the elements from the tail and those arising from the currently processed step may be connected by newly generated arcs.

We therefore introduce an auxiliary data structure that keeps information about the tails of intermediate diagrams. For every action \( a \in \Sigma \), the data structure consists of an ordered dependence list \( \text{ORD}_a \), followed by two sets of visible sources, \( \text{SVS}_a \) and \( \text{WVS}_a \), giving strongly visible sources and properly weakly visible sources of \( a \), as well as a pointer to the latest occurrence of \( a \).

The lists and sets have the size at most \( k \). During the computation, we also use temporary copies of sets of sources (visibility sets).

In contrast to the case of Mazurkiewicz traces, when constructing Hasse diagrams of comtraces, we have to deal not only with two kinds of dependencies between action occurrences, but also with their local contexts. We say that occurrences \( \alpha \) and \( \beta \) are \textit{locally dependent} if their folded actions are not completely independent by which we mean that \( (\ell(\langle \alpha \rangle) \times \ell(\langle \beta \rangle)) \cap \text{ind}^{-1} \neq \emptyset \). The list \( \text{ORD}_a \) contains all the actions whose latest occurrences are locally dependent with latest occurrence of \( a \).

The set \( \text{SVS}_a \) contains all the actions \( b \) whose latest occurrences were \textit{strongly visible} from the latest occurrence labelled with \( a \), or are in the same folded action as the latest occurrence of \( a \). In other words, there exists a strong path from \( \beta \) to \( \alpha \), where \( \beta \) and \( \alpha \) are respectively the latest occurrences of actions \( b \) and \( a \), or \( \langle \alpha \rangle = \langle \beta \rangle \). Each such \( \beta \) is called a \textit{strong source} of \( \alpha \).

Similarly, the set \( \text{WVS}_a \) contains all the actions \( b \) whose latest occurrences were weakly visible from the latest vertex labelled with \( a \). In other words, there exists a weak path from \( \beta \) to \( \alpha \), where \( \beta \) and \( \alpha \) are respectively the latest occurrences of actions \( b \) and \( a \) in the diagram constructed so far. Each such \( \beta \) is called a \textit{weak source} of \( \alpha \).

The last element in the data structure, \( \text{LST}_a \), is a pointer to the most recent vertex labelled with the action \( a \) in the diagram being constructed.

Before generating the Hasse diagram of \( v \), we set all the pointers to \textit{null}, and make all the sets of sources and all dependence lists empty. It is worth stressing that dependence lists are computed dynamically and some actions present in the list \( \text{ORD}_a \) may be even independent with action \( a \).

We then process consecutive steps of the input sequence \( v \), updating the auxiliary data structure (possibly many times during a single phase), and creating
new vertices and arcs. We will now describe a single phase of the algorithm which starts with the Hasse diagram \( H(w) = (\text{occ}(w), \preceq^h_w, \succeq^h_w) \) of a step sequence \( w \) of length \( m \) together with an auxiliary data structure for its tail \( \text{tail}(w) \). To construct the Hasse diagram of \( wA \), for \( A \in S \), we proceed in two stages.

**Example 15.** Consider the constrate alphabet form Example 4, a step sequence \( w = \{d\}\{a, b\}\{c\}\{d\} \) and a step \( A = \{a, b, c\} \). Then the Hasse diagram of \( H(w) \) together with an auxiliary data structure as well as the set \( A \) look as follows:

\[
\begin{array}{c}
\text{ORD} \\
\text{SVS} \\
\text{WVS} \\
\text{LST}
\end{array}
\begin{array}{c|c|c|c}
 a & (d, c, a) & \{a\} & \emptyset & \rightarrow a_1 \\
b & (d, c, b) & \{b\} & \emptyset & \rightarrow b_1 \\
c & (d, c, b, a) & \{a, c\} & \{b\} & \rightarrow c_1 \\
d & (d, c, a, b) & \{a, b, c, d\} & \emptyset & \rightarrow d_2 \\
\end{array}
\]

\( \Box \)

In the first stage, we identify [2] all the strongly connected components \( \text{SCC} \) of the directed graph \((\text{occ}_{m+1}(wA)), \preceq^h_{wA}\cap_{wA} \text{occ}_{m+1}(wA))\) which gives us a partition of \( \text{occ}_{m+1}(wA) \) into folded actions (see Proposition 5). We use them to produce new vertices. In this way, we construct the missing part of \( \text{tail}(wA) \). According to Proposition 5, the directed graph \( \text{DAG} = (\text{SCC}, \preceq_{\text{SCC}}) \) is a poset. We compute any of its linearisations using the topological sort [2] and store the results in the list \( TS = (TS_1, TS_2, \ldots, TS_q) \). This is crucial for the correctness of whole procedure. Let us just notice that each \( [wTS_1TS_2\ldots TS_k] \), for \( 1 \leq k \leq q \), is a correct prefix of \( [wA] \) (in other words, \( A = TS_1 \ldots TS_k \circ TS_{k+1} \ldots TS_q \)).

**Example 16.** Continuing Example 15, after the first stage we obtain:

\[
\begin{array}{c}
\text{ORD} \\
\text{SVS} \\
\text{WVS} \\
\text{LST}
\end{array}
\begin{array}{c|c|c|c}
 a & (a_2, c_2) & \{a_2, c_2\} & \emptyset & \rightarrow a_1 \\
b & (b_2) & \emptyset & \rightarrow b_1 \\
c & (c_1) & \emptyset & \rightarrow c_1 \\
d & (d_1) & \emptyset & \rightarrow d_1 \\
\end{array}
\]

\( \Box \)

In the second stage, we scan the list \( TS \) and add new arcs. We divide the processing of each vertex \( \Delta = TS_i \) into two parts. Firstly, we check the necessity of adding new arcs from \( \text{tail} \) to \( \Delta \). Secondly, we update the structure of visibility sets and dependence lists. It is important to emphasize that the structure of the diagram and its tail is updated many times in the second stage of each phase.
At the beginning of the first part we initialise two auxiliary sets SVS and WVS with the value \( \emptyset \). To check the necessity of adding an arc from the \( \text{tail} \) to vertex \( \Delta \), we need to compute the dependence list for this vertex. Notice that all linear orders stored in dependence lists \( ORD_a \) are compatible and it is possible to compute a minimal linear order that contains them. We do this in an arbitrary way producing a single list \( ORD_\Delta \). After that we scan the list \( ORD_\Delta \). For each \( b \) from \( ORD_\Delta \), we check whether the pointer \( LST_b \) is not \( \text{null} \). If it is not, there are two possibilities, as we can try to add new strong arc or new weak arc. To check the type of local dependence between folded action \( \Delta = (\alpha) \) and the latest occurrence of \( b, \beta \), we have to fold action occurrence \( \beta \) into folded action \( (\beta) \) as well. We do that by checking all outgoing arcs. All occurrences \( \delta \) connected with a short loop \( (\beta \subset \gamma \subset \beta) \) are in the same vertex of folded action as \( \beta \). Now, if \((\ell(\Delta) \times \ell(\langle \beta \rangle)) \cap (\text{wdp} \cup \text{dep} \cup \text{ssm}) = \emptyset \) then these folded actions are strongly dependent \((\langle \beta \rangle \prec h \Delta)\). Otherwise they are weakly dependent \((\langle \beta \rangle \preceq h \Delta)\).

In the first case (trying to add a strong arc), we check if \( \ell(\langle \beta \rangle) \cap \text{SVS} = \emptyset \). If so, we add a new strong arc from \( \langle \beta \rangle \) to \( \Delta \) and update the sets \( \text{SVS} \) and \( \text{WVS} \) by adding \( \text{SVS}_b \cup \text{WVS}_b \) to \( \text{SVS} \) and subtracting \( \text{SVS} \) from \( \text{WVS} \). In the second case (trying to add a weak arc), we have to check if \( \ell(\langle \beta \rangle) \cap (\text{SVS} \cup \text{WVS}) = \emptyset \). If so, we add a new weak arc from \( \langle \beta \rangle \) to \( \Delta \) and update the set \( \text{SVS} \) and \( \text{WVS} \) by adding \( \text{SVS}_b \setminus \ell(\langle \beta \rangle) \) to \( \text{SVS} \) and \( \text{WVS}_b \cup \ell(\langle \beta \rangle) \) to \( \text{WVS} \). To preserve the separation of the sets \( \text{SVS} \) and \( \text{WVS} \), we also subtract \( \text{SVS} \) from \( \text{WVS} \). At the end, we split the vertex \( \Delta \) into \(|\Delta| \) vertices (splitting also all newly added arcs), and make from the resulting set of new vertices a weakly connected clique.

**Example 17.** Continuing Example 16, during the second stage we have:

\[
\begin{array}{c}
\text{ORD} \\
\{a, c, d\} \\
\{b\} \\
\emptyset \\
\{d\}
\end{array}
\begin{array}{c}
\text{SVS} \\
\{a, c, d\} \\
\{b\} \\
\emptyset \\
\{d\}
\end{array}
\begin{array}{c}
\text{WVS} \\
\emptyset
\end{array}
\begin{array}{c}
\text{LST} \\
a \rightarrow a_2 \\
b \rightarrow b_2 \\
c \rightarrow c_2 \\
d \rightarrow d_2
\end{array}
\]

After the second stage we obtain:
At this point we have added all the necessary arcs, but the tail of the diagram has changed and therefore the visibility sets should also be updated. The second part starts, by subtracting the labels \( \ell(\Delta) \) from every visibility set \( SVS_a \) and \( WVS_a \). Then we update all the lists \( ORD_a \) by moving, one by one, the labels of the elements of the folded action \( \Delta \) to the beginning of the list or removing them if their latest occurrences are no more locally dependent with the latest occurrences of labels of \( \Delta \). These updates can be done in arbitrary but fixed order of labels of the action occurrences belonging to \( \Delta \). The last operation is the updating of the visibility sets for the labels of the elements of \( \Delta \). For each \( a \) that belongs to \( \ell(\Delta) \), we set \( SVS_a \) to \( SVS \cup \ell(\Delta) \), and \( WVS_a \) to \( WVS \cup \ell(\Delta) \). We also update (for all \( a \in \ell(\Delta) \)) the values of the pointers \( LST_a \).

Example 18. Continuing Example 17, after adding the next step \{c\} we get:

![Diagram](image)

### Algorithm 1: Hasse diagram

**INPUT:** step sequence \( v = A_1 \ldots A_q \) over comtrace alphabet \( (\Sigma, \text{sim}, \text{ser}) \)

**OUTPUT:** Hasse diagram of \( H(v) \)

1. **for all** \( a \in \Sigma \) **do**
2. \( LST_a := -1; SVS_a := \emptyset \) \( WVS_a := \emptyset \)
3. **end for**
4. Compute relations \( \text{dep}, \text{ind}, \text{ssm}, \text{wdp} \)
5. Compute sets \( ORD_a \) using relation \( \text{ind} \)
6. **for** \( i := 1 \) **to** \( q \) **do**
7. Compute all strongly connected components \( SCC \) of \( \{\text{occ}(A_1 \ldots A_i), \text{occ}(A_1 \ldots A_i) / \text{occ}(A_1 \ldots A_i) \times \text{occ}(A_1 \ldots A_i)\} \)
8. Compute list \( TS \) using topological sorting on \( \text{DAG} = (\text{SCC}, \supseteq) \times (\text{SCC} \times \text{SCC}) \)
9. **for all** \( \Delta \in TS \) **do**
10. Add new strong and weak arcs \{Part 1\}
11. Update data structure \{Part 2\}
12. **end for**
13. **end for**

### Algorithm 2: Stage 2, Part 1: Add new arcs

1. Add vertex \( \Delta \)
2. \( SVS = WVS = \emptyset \)
3. Combine lists \( ORD_a \) for all actions \( a \in \ell(\Delta) \) into \( ORD_\Delta \)
4. **for all** \( b \in ORD_\Delta \) **do**
5: \( \beta = LST_b \)
6: Compute \( \langle \beta \rangle \)
7: if \( \langle \beta \rangle \) is strongly dependent with \( \Delta \) and \( \langle \beta \rangle \cap SVS = \emptyset \) then
8:    Add a strong arc \( \langle \langle \beta \rangle, \Delta \rangle \)\)
9:    \( SVS := SVS \cup SVS_b \cup WVS_b \)
10:   \( WVS := WVS \setminus SVS \)
11: end if
12: if \( \langle \beta \rangle \) is weakly dependent with \( \Delta \) and \( \langle \beta \rangle \cap (SVS \cup WVS) = \emptyset \) then
13:    Add a weak arc \( \langle \langle \beta \rangle, \Delta \rangle \)\)
14:    \( SVS := SVS \cup (SVS_b \setminus \ell(\langle \beta \rangle)) \)
15:   \( WVS := (WVS \cup WVS_b) \setminus SVS \)
16: end if
17: end for
18: for all \( \alpha \in \Delta \) do
19:    Add vertex \( \alpha \)
20:    for all arc \( \langle \Delta', \Delta \rangle \) do
21:      for all \( \beta \in \Delta' \) do
22:        Add arc \( \langle \beta, \alpha \rangle \)
23:      end for
24:    end for
25: end for
26: Remove vertex \( \Delta \) (with all arcs)

---

**Algorithm 3: Stage 2, Part 2: Update the data structure**

1: for all \( b \in \Sigma \) do
2:    \( SVS_b := SVS_b \cup \ell(\Delta) \)
3:    \( WVS_b := WVS_b \cup \ell(\Delta) \)
4:    for all \( a \in \ell(\Delta) \) do
5:      Move \( a \) to the head of \( ORD_b \)
6:    end for
7: end for
8: for all \( a \in \ell(\Delta) \) do
9:    \( SVS_a := SVS \setminus \ell(\Delta) \)
10:   \( WVS_a := WVS \setminus \ell(\Delta) \)
11: Update \( LST_a \)
12: end for

We will now discuss the complexity of the proposed algorithm. In what follows, \( d \) denote the size of currently processed fold, \( k \) denotes the size of the alphabet \( \Sigma \), and \( n \) denotes the size \( |occ(v)| \) of the input step sequence \( v \).

The algorithm is on-line, and so we do not have to store the entire step sequence nor the Hasse diagram. All we need to maintain is the structure that describes the tail of the current diagram. In this structure we store \( k \) lists \( ORD_a \) of size \( k \) each, which contributes \( O(k^3) \) to the memory complexity. We also store
$k$ sets $S_{VS_a}$, $k$ sets $W_{VS_a}$, $k$ pointers $L_{ST_a}$, and a constant number of other variables, but these do not increase the memory complexity.

Estimating time complexity is more involved. Let us start by evaluating the two parts of the second stage (see Algorithm 2 and Algorithm 3). In the first part we combine $dk$ lists of size $k$ while preserving the dependencies stored in these lists (line 3). It gives $O(dk^2)$ time complexity. After that, for every action $b \in Ord_{\Delta}$ we attempt to add some arcs. The tests carried out in lines 7 and 11 gives time complexity $O(dk)$. The set operations in lines 9, 10, 13 and 14 and computing $(\beta)$ in line 6 give $O(k)$ time complexity each, while the time involved in adding an actual arc is constant. The whole loop (lines 4-14) has therefore time complexity of $O(dk^2)$. The time complexity of the second loop (lines 15-19), and of removing a vertex in line 20 are clearly $O(dk^2)$. As a result, we have $O(dk^2)$ total time complexity for the first part of the second phase.

The second part has also complexity of $O(dk^2)$. In Algorithm 3, we have two loops with at most $k$ iterations. In each iteration, we carry out some set operations of the complexity $O(k)$ each, and $d$ rearrangements of lists $ORD_a$ which can also be done in $O(dk)$ time.

Let us now calculate the total time complexity of Algorithm 1. The preprocessing phase (lines 1-5) has the time complexity of $O(k^2)$. Then we have a main loop with at most $n$ iterations. The first phase involves some operations on graphs of the size $O(k^2)$ or $O(k)$ (by the size of a graph we mean its total number of vertices and arcs). Detecting strongly connected components and topological sorting of a directed acyclic graph are both linear in its size (see [14]), and so we can carry out the first phase for each step in $O(k^2)$ time. This contributes $O(nk^2)$ component to the overall time complexity.

Finally, we notice that the number of the iterations in loop from line 9 is linear in the size of the original step sequence. Moreover, every folded action is counted only once, and so when calculating the total complexity, we can skip the $d$ factor of the complexity of the second stage. This contributes another $O(nk^2)$ component to the overall time complexity. Hence the time complexity is equal to $O(nk^2)$. As in the case of traces, $k$ is bounded by $|\Sigma|$ (which is fixed for a given system) and much smaller than $n$. Hence, the above algorithm can in practice be considered as linear in the size of its input, and so optimal.

6 Applications

A major advantage of Hasse diagrams of comtraces and the data structure that is used by the algorithm described in the previous section is a convenient and efficient representation of comtraces. In fact, it is worth adding information about the head of a comtrace. In the extended structure, we store pointers to all the elements belonging to $head(w)$. Intuitively, when concatenating two comtraces, all new connections between their folded Hasse diagrams would in fact be between the tail of first diagram and the head of the second one.

Another application of Hasse diagrams is a method of comparing comtraces as two step sequences, $w$ and $v$, belong to the same comtrace if and only if their
Hasse diagrams are equal. More formally:

\[ w \equiv_{\Theta} v \iff H(w) = H(v) \iff \bar{H}(w) = \bar{H}(v) \, . \]

Testing for equality of two graphs is linear in their size, and so using Hasse diagrams allows testing for contrace equivalence in \( O(nk^2) \) time.

Using Hasse diagram \( H(w) = (\text{occ}(w), \preceq^h, \preceq^b) \) with the head structure \( \text{head}(w) \), we can also compute the Foata normal form of \([w]\). To do so, we only need to compute the maximal step \( F_1 \) containing the head. The step \( F_1 \) is the first step of the new step sequence \( F \). After that, we proceed by computing the set \( F_2' \) of all action occurrences that are the targets of the strong arcs originating in \( F_1 \).

To compute the second step \( F_2 \), we need to compute the transitive closure of \( F_2' \) (closing the relation \( \preceq^b \)), testing that none of the vertices contained in \( F_2 \) is in the relation \( \preceq^b \) with \( F_2' \). The correctness of the resulting procedure follows directly from the definition of Foata normal form. Its time complexity is linearly dependent on the size of Hasse diagram and therefore equal to \( O(nk) \).

7 Conclusions

In this paper, we presented an efficient way of generating graph theoretic representations of contraces. We also provided a number of properties of folded stratified order structures.

In our future work we plan extend our current results to cover also generalised contraces and generalised so-structures [5,6]. Another direction is the development of more efficient algorithm for checking equivalence of two step sequences. Yet another, arguably most challenging, problem is to use Hasse diagrams of contraces to define an algebra of contraces with a suitable iteration operator.

References