

Maximal Confluent Processes

Xu Wang

International Institute of Software Technology, United Nations University
PO Box 3058, Macau
wx@iist.unu.edu

Abstract. In process semantics of Petri Net, a non-sequential process is a concurrent run of the system represented in a partial order-like structure. For transition systems we can define a similar notion of concurrent run by utilising the idea of confluence. Basically a confluent process is an acyclic confluent transition system that is a *partial unfolding* of the original system. Given a non-confluent transition system G , how to find maximal confluent processes of G is a theoretical problem having many practical applications.

In this paper we propose an unfolding procedure for extracting maximal confluent processes from transition systems. The key technique we utilise in the procedure is the construction of *granular configuration structures* (i.e. a form of event structures) based on diamond-structure information inside transition systems.

1 Introduction

Confluence is an important notion of transition systems. Previously there has been extensive work devoted to its study, e.g. [12, 9, 8, 11]. In [9] confluence is studied from the perspective of non-interleaving models, where it was concluded that in order to characterise the class of confluent transition systems the underlying event-based models needs to support the notion of *or-causality* [19, 22].

In this paper we are going to study the idea of maximal confluent sub-systems of a non-confluent transition system, also from a non-interleaving perspective. It can be regarded as an extension of the notion of non-sequential pocesses in Petri Net [7, 3, 4] onto transition systems. We call it *maximal confluent process* (MCP). Intuitively a maximal confluent process is a concurrent run of the system that is maximal both in length and in degree of concurrency. A non-confluent system has multiple such runs. Non-maximal concurrent runs can be deduced from maximal ones, e.g. by restricting concurrency (i.e. strengthening causality relation).

Like non-sequential processes, which can be bundled together to form *branching processes* of Petri Net, the set of maximal confluent processes (extracted from a given transition system) can coalesce into a *MCP branching processes* of the original system. Such branching processes record, in addition to causality information, also the ‘choice points’ of the system at which different runs split from each other. In a non-interleaving setting the ‘choice points’ are formalised as

(immediate) *conflicts* on events. The arity of the conflicts can be *non-binary*, thus giving rise to the so called *finite conflicts*. For instance, in state s_0 of Figure 1 actions a , b and c form a ternary conflict, which induces the three maximal concurrent runs of the system (i.e. the three subgraphs on the right).

In this paper we propose an unfolding procedure to construct *granular configuration structures* from transition systems. The procedure preserves the maximality of confluence in such a way that each generated configuration corresponds to a prefix of some maximal concurrent run. Configuration structures are an event structure represented in a global-state based fashion [18, 17]. They support or-causality as well as finite conflicts.

2 Motivating Examples

We first look at two examples in order to build up some intuitions for maximal confluent processes.

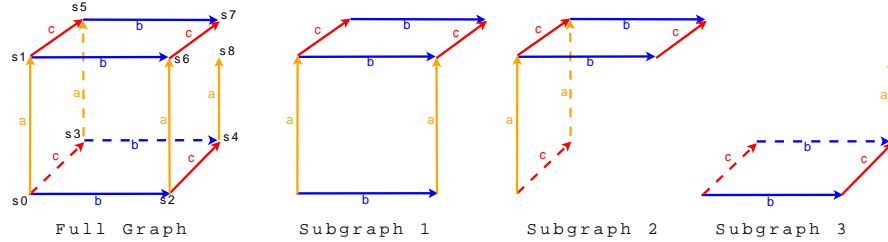


Fig. 1. A running example

The first example is the left-most graph in Figure 1, which is an LTS in the shape of a *broken cube* (i.e. replacing transition $s_4 \xrightarrow{s} s_8$ by $s_4 \xrightarrow{s} s_7$ will give rise to a true cube-shaped LTS). The three subgraphs on its right are confluent subgraphs of the broken cube. Moreover, they are maximal such subgraphs; adding any state or transition to them will invalidate their confluence. They are exactly the maximal confluent processes we are looking for.

For the general cases, however, maximal confluent processes do not coincide with maximal confluent subgraphs. Let us look at the left-most LTS in Figure 2. The maximal confluent subgraphs of such system are the four maximal simple paths in the graph, i.e. $s_1 \xrightarrow{a} s_2 \xrightarrow{a} s_3$, $s_1 \xrightarrow{a} s_2 \xrightarrow{b} s_3$, $s_1 \xrightarrow{b} s_2 \xrightarrow{a} s_3$, and $s_1 \xrightarrow{b} s_2 \xrightarrow{b} s_3$. But its maximal confluent processes have three members, MCP 1-3 in Figure 2. Two subgraphs $s_1 \xrightarrow{a} s_2 \xrightarrow{b} s_3$ and $s_1 \xrightarrow{b} s_2 \xrightarrow{a} s_3$ are combined into one process, MCP 1. MCP 1 is not a subgraph because state s_2 of the original LTS is split into two states.

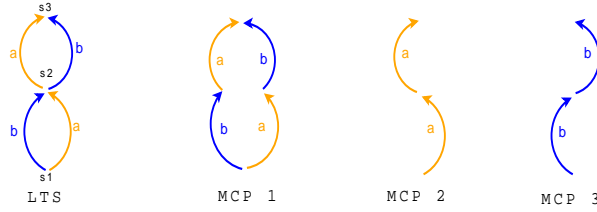


Fig. 2. The second example

The idea of maximal confluent processes has interesting applications. The extraction of maximal confluent processes from a given transition system can be regarded as a deep form of commutativity analysis on the system, which is fully dynamic (i.e. state-dependent) and global (i.e. checking infinite number of steps into future). For instance, they can be used in partial order reduction [13, 6, 15] to define a canonical notion of optimal reduction, *weak knot* [10]. The challenge, however, lies in how to find a procedure that uses only *local* diamond-structure information inside transition systems to extract maximal confluent processes.

Now let us develop a formal framework to study the problem.

3 Maximal Confluent Processes

Definition 1. A transition system (TS) is a 4-tuple $(S, \Sigma, \Delta, \hat{s})$ where

- S is a set of states,
- Σ is a finite set of actions (ranged over by a, b , etc.),
- Δ is a partial function from $S \times \Sigma$ to S (i.e. the transition function)¹, and
- $\hat{s} \in S$ is the initial state.

Fix a TS, $G = (S, \Sigma, \Delta, \hat{s})$, and define:

- a transition $t = s \xrightarrow{a} s'$ means $(s, a, s') \in \Delta$;
- a consecutive sequence of transitions $L = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \dots s_{n-1} \xrightarrow{a_n} s_n$ means $s_{i-1} \xrightarrow{a_i} s_i$ for all $1 \leq i \leq n$. L is called an *execution* (i.e. sequential run) of G from s_0 to s_n producing *trace* $a_1 \dots a_n$. When $s_0 = \hat{s}$ we further call it a *system execution* of G ; and we use $\mathcal{L}(G)$ to denote the set of system executions of G .

¹ Note that our transition systems are actually deterministic transition systems in the classical sense. It gives us simplicity in theory presentation while at the same time sacrificing few technical insights. Our theory will be easily extendable onto *general transition systems* (which include classical non-deterministic transition systems) by adding one more layer of re-labelling, i.e. at places actions or labels are used we use transitions instead. Furthermore, note that most notations developed in this section also apply to general transition systems.

- $s \xrightarrow{a_1 \cdots a_n} s'$ means there exists an execution of G from s_0 to s_n producing trace $a_1 \cdots a_n$;
- $s \xrightarrow{a}$ means $\exists s' : s \xrightarrow{a} s'$, while $s \not\xrightarrow{a}$ means $\neg s \xrightarrow{a}$;
- $eb(s)$ denotes the set of actions enabled at s , i.e. $\{a \mid s \xrightarrow{a}\}$;
- $Reach(s)$ denotes the set of states reachable from s ;
- given any $s \in Reach(\hat{s})$, $G/s = (S, \Sigma, \Delta, s)$ denotes the new transition system generated after the evolution to s ;
- and if G is acyclic, we further define:
 - $s \sqsubseteq s'$ means $s' \in Reach(s)$, i.e. s' is a *subsequent* state of s (or s is an *earlier* state of s');
 - given any $X \subseteq S$, $min(X)$ denotes the set of \sqsubseteq -minimal states inside X , while X^\downarrow denotes the \sqsubseteq -downward closure of X ;
 - $s \parallel_{\sqsubseteq} s'$ means s and s' are incomparable w.r.t. \sqsubseteq ;
 - and given any $s \in Reach(\hat{s})$, s/G denotes the restriction of G to $\{s\}^\downarrow$, i.e. $s/G = (\{s\}^\downarrow, \Sigma, \{(s_0, a, s_1) \mid (s_0, a, s_1) \in \Delta \wedge s_0, s_1 \in \{s\}^\downarrow\}, \hat{s})$.

When there is any danger of confusion, we use \rightarrow_G to say the transitions come from a TS named G . Similarly we use S_G for the set of states and \hat{s}_G for the initial state of G .

TSes can be related to each other by *partial unfolding* relation:

- We say G is a partial unfolding of G' if there exists a function f from S_G to $S_{G'}$ such that $f(\hat{s}_G) = \hat{s}_{G'}$ and $s \xrightarrow{a}_G s' \implies f(s) \xrightarrow{a}_{G'} f(s')$.

As its name suggests partial unfolding unwinds just part of a transition system. When f is injective, partial unfolding is reduced to be *subgraph* relation. In the rest of the paper, whenever the homomorphism f of any subgraph relation is left unspecified, we assume f is the *identity function*.

Of course, we can also fully unwind TSes, giving rise to the *unfolding* relation:

- We say G is an unfolding of G' if G is a partial unfolding of G' and, for all system execution $\hat{s}_{G'} \xrightarrow{a_1}_{G'} s'_1 \xrightarrow{a_2}_{G'} s'_2 \cdots s'_{n-1} \xrightarrow{a_n}_{G'} s'_n$ of G' , there is a system execution $\hat{s}_G \xrightarrow{a_1}_G s_1 \xrightarrow{a_2}_G s_2 \cdots s_{n-1} \xrightarrow{a_n}_G s_n$ of G s.t. $s'_i = f(s_i)$ for all $1 \leq i \leq n$.

Of all TSes, a particular interesting subclass of TSes is *confluent TSes*.

- G is confluent if, for all $s \in S_G$ and $a, b \in eb_G(s)$ (with $a \neq b$), a and b form a local diamond at s , i.e. $\exists s_3 \in S_G : s \xrightarrow{ab}_G s_3 \wedge s \xrightarrow{ba}_G s_3$.

In the rest of the paper we use $a \diamond_s b$ to denote a diamond local at s and built from a and b actions. The notation can be extended to multi-dimension diamonds. We use $\diamond_s A$ to denote a n -dimension (where $n = |A|$) diamond local at s and built from members of A , i.e. given any $B \subseteq A$, there exists a unique $s' \in S$ such that $s \xrightarrow{a_1 \cdots a_m} s'$ for all permutation $a_1 \cdots a_m$ of B .

It is interesting to note that all local diamonds inside a partial unfolding are inherited from those of the original TS. They are the unwinded versions of

the original diamonds (c.f. MCP 1 in Figure 2). Furthermore, since a partial unfolding can visit a state of the original TS more than once (esp. when the original TS is cyclic), we can choose to unwind a different diamond on subsequent visits to the state.

Now we are ready to define the notion of concurrent runs of TSes:

- We say an acyclic confluent TS F is a *confluent process* of G if F is a partial unfolding of G .

A confluent process F can be finite or infinite. For a finite confluent process F , it has a unique maximal state, denoted \check{s}_F .

Define $ms()$ to be the function that converts a sequence of actions into a multiset of actions, i.e. $\mathcal{A} = ms(\alpha)$ iff $\mathcal{A}(a)$ gives the number of times a occurs in α for all $a \in \Sigma$. Use $+$ as multiset union. Then some properties of confluent processes are:

Lemma 1. *Given any state s in a confluent process F , $\hat{s}_F \xrightarrow{\alpha}_F s$ and $\hat{s}_F \xrightarrow{\alpha'}_F s$ implies $ms(\alpha) = ms(\alpha')$. Thus we can also use the notation $\hat{s}_F \xrightarrow{ms(\alpha)}_F s$.*

Lemma 2. *Given any $\hat{s}_F \xrightarrow{A_1}_F s$ and $\hat{s}_F \xrightarrow{A_2}_F s'$, there exists a state s'' in F s.t. $\hat{s}_F \xrightarrow{A_1+A_2}_F s''$.*

- We say a confluent process F of G is a *maximal confluent process* (MCPs) if F is maximal w.r.t. partial unfolding relation, i.e. F is a partial unfolding of another confluent process F' implies F' and F are isomorphic².

When restricted to confluent processes, partial unfolding relation is reduced to subgraph relation (c.f. the lemma below). Thus MCPs are ‘maximal confluent subgraphs’. In addition, there is a unique minimal confluent subgraph of G , denoted \hat{G} . \hat{G} is the trivial TS with a single state and empty transition function, i.e. $\hat{G} = (\{\hat{s}_G\}, \Sigma, \{\}, \hat{s}_G)$.

Lemma 3. *Given two confluent processes F and F' of G , F is a partial unfolding of F' implies the homomorphism f between F and F' is injective.*

Proof. Assume f is not injective. Then there exists $s' \in S_{F'}$ such that $\exists s_1, s_2 \in S_F : f(s_1) = f(s_2) = s'$. Due to acyclicity of F' , $s_1 \parallel_{\square} s_2$ is true, but we can find some $s_0 \in S_F$ such that s_0 can reach s_1 by trace α and s_2 by trace β . Since from $f(s_0)$ both α and β will lead to s' in acyclic confluent F' , α and β are permutation of each other. Thus from s_0 , α and β will lead to the same state in F (due to its confluence). Contradiction with the $s_1 \neq s_2$.

In the rest of the paper we will use \preceq to denote subgraph relation on confluent processes. Relation \preceq allows a confluent process to be reduced in two different

² In the rest of the paper we will freely use F to denote an acyclic confluent graph or to denote its isomorphism class.

dimensions: the degree of concurrency and the length of causality chains. Thus MCPs represent the longest possible runs of the system in a maximally concurrent fashion. In a transition system with cycles that implies MCPs are often infinite graphs: finite MCPs are those derived from terminating runs (i.e. ending in a state where there is no outgoing transitions).

More refined relations on confluent processes that reduces only one of the dimensions can also be defined:

- Given two confluent processes F and F' , we say F is a (concurrency) *tightening* of F' (or F' is a *relaxation* of F), denoted $F \preceq_r F'$, if F is a subgraph of F' and, for all $s \in S_F$ and $a \in eb_{F'}(s)$, there exists a subsequent state $s' \in S_F$ of s s.t. $a \in eb_F(s')$.
- Given two confluent processes F and F' of G , we say F is a *prefix* of F' (or F' is an *elongation* of F), denoted $F \preceq_e F'$, if F is a subgraph of F' and there exists a function p from S_F to 2^{Σ} (i.e. *the pending action function*) s.t. $s \in S_F \implies p(s) = eb_{F'}(s) \setminus eb_F(s)$ and $s \xrightarrow{a}_F s' \implies p(s) \subseteq p(s')$.

Subgraph relation is decomposable into the two refined relations.

Lemma 4. $F \preceq F''$ 1) iff there exists some F' s.t. $F \preceq_r F' \preceq_e F''$ and 2) iff there exists some F' s.t. $F \preceq_e F' \preceq_r F''$.

The intuition behind the refined relations can better be understood using the notion of ‘events’.

- Given a confluent process F , we say a state $s \in S_F$ is the *origin* of an action occurrence, say a , if $a \in eb_F(s)$ and $s_0 \xrightarrow{b}_F s \implies a = b \vee a \notin eb_F(s_0)$.
- An occurrence of a with origin s gives rise to a *granular event*, denoted T , which is the set of a -transition reachable from s by firing only non- a transitions in F . Conversely, given T we use $lb_F(T)$ and $o_F(T)$ to denote its label a and origin s resp.
- Given two confluent processes $F \preceq F'$ and two granular events T in F and T' in F' , we say T and T' are the *same event* if $T \subseteq T'$ and $o_F(T) = o_{F'}(T')$; and we say T is a *postponed occurrence* of T' if $T \subseteq T'$ and $o_F(T) \neq o_{F'}(T')$.
- We say two granular events T and T' of F are *or-causally coupled* if $T \cap T' \neq \{\}$.

The or-causal coupling relation is reflexive and symmetric. Its transitive closure, which is an equivalence relation, can be used to partition the set of granular events in F . That is, each equivalence class \mathcal{E} gives rise an *event* $T = \bigcup_{T_0 \in \mathcal{E}} T_0$. Note that an event does not have a unique origin; thus we replace $o_F(T)$ by $O_F(T)$ to denote its set of origins.

- Given two confluent processes $F \preceq F'$ and two events T in F and T' in F' , we say T and T' are the *same event* if $T \subseteq T'$ and $O_F(T) = O_{F'}(T') \cap S_F$, and we say T is a *delayed occurrence* of T' if $T \subseteq T'$ and $O_F(T) \neq O_{F'}(T') \cap S_F$.

Based on the notions of events we can see that tightening on F' *delays* (but not removes) events in F' while prefixing on F' *removes* (but not delays) events inside F' .

One fact noteworthy is that, as we elongate a confluent process, events can become ‘enlarged’ through the addition of new granular events (even though they remain the same events). However, this addition has an upper-limit as the ‘size’ of an event will eventually stabilise.

Lemma 5. *Given a strictly increasing (w.r.t. \preceq_E) infinite sequence of confluent processes $F_0 F_1 \dots F_i \dots$, T is an event in F_i implies there exists some $j \geq i$ and event T' in F_j s.t. T and T' are the same event and T' is stabilised at j , i.e. for any $n \geq j$, T'' of F_n is the same event as T' implies $O_{F_j}(T') = O_{F_n}(T'')$.*

Some further facts about the refined relations are:

Lemma 6. *Given two finite confluent processes $F \preceq F'$, we have 1) $F \preceq_r F'$ iff $\check{s}_F = \check{s}_{F'}$, and 2) $F \preceq_e F'$ iff $F = \check{s}_F/F'$.*

Proof. Obvious for $F \preceq_r F'$.

Right to left for $F \preceq_e F'$: Define $p(s_F) = eb_{F'}(s_F) \setminus eb_F(s_F)$ for all $s_F \in S_F$. If $p(s) \not\subseteq p(s')$ for some $s \xrightarrow{a}_F s'$, then for all $b \in p(s) \setminus p(s')$ we have $\Delta'_F(s', b) \in S_F$ but $\Delta'_F(s, b) \notin S_F$. Contradiction with $F = \check{s}_F/F'$ since $\Delta_{F'}(s', b) \in Reach_{F'}(\Delta_{F'}(s, b))$ and $\check{s}_F \in Reach_{F'}(\Delta_{F'}(s', b))$.

Left to right for $F \preceq_e F'$: Let $p()$ be the pending action function and $s \in S_{F'} \setminus S_F$ be a minimal (i.e. earliest) state s.t. $\check{s}_F \in Reach_{F'}(s)$. Then $s_0 \xrightarrow{a}_{F'} s$ implies $s_0 \in S_F$ and $a \in p(s_0)$. Thus no path connecting s_0 to \check{s}_F in F will execute a action. However, in F' there is a path connecting s_0 and \check{s}_F and executing a . Contradiction with F' being a confluent process.

In another word the set of prefixes of F' corresponds 1-1 to the set of states of F' .

- A confluent process F is said to be a *maximally relaxed process* (i.e. MRP) if F is maximal w.r.t. \preceq_r .
- A confluent process F is said to be a *maximally elongated process* (i.e. MEP) if F is maximal w.r.t. \preceq_e .
- A confluent process F is said to be an *MCP prefix* if there exists an MCP F' s.t. F is a prefix of F' . MCP prefixes are the initial parts of complete maximally-concurrent runs.

Naturally one can imagine that MCPs are generated step by step by unfolding local diamonds in the states it visits; MCPs usually prefers to unfold larger diamonds in each step. However, the maximality of MCPs, unlike diamonds, is a global property. Sometimes choosing a strictly smaller diamond to unfold at an early state might lead to a larger diamond in subsequent states. This phenomenon is similar to the phenomenon of *confusion* in Petri Net.

- Given a state $s \in S_G$, we say $A \subseteq \Sigma$ is a *maximal concurrent step* (MCS) at s if A is a maximal diamond local at s .

- Given a state $s \in S_G$, we say $A \subseteq \Sigma$ is an *MCP step* (MPS) at s if there exists a MCP F of G s.t. $\exists s_F \in S_F : f(s_F) = s \wedge eb_F(s_F) = A$.

Given a state s , the set of its MPSes are not necessarily downward closed or mutually incomparable (w.r.t subsethood). As an example, imagine an event structure with four events $e1, e2, e3$ and $e4$ labelled by action a, b, c and d resp. $e3$ and $e4$ causally depends on $e2$ while $e3$ is in conflict with $e1$. In the transition system generated by the event structure, a and b form a maximal diamond at the initial state. However, taking the $a \diamond b$ diamond will destroy the future $c \diamond d$ diamond which is reachable by taking the b action only. Thus $\{a, b\}$ and $\{b\}$ are both MPSes at the initial state whilst $\{a\}$ is not.

Similarly we can see that not all MCP prefixes are MRPs, even though all MRPs are MCP prefixes:

Lemma 7. *A confluent process F is an MRP implies F is an MCP prefix.*

Proof. Assume F' is an MCP s.t. F is a subgraph of F' . Define $p(s_F) = eb_{F'}(s_F) \setminus eb_F(s_F)$ for all $s_F \in S_F$. If $p(s) \not\subseteq p(s')$ for some $s \xrightarrow{a}_F s'$, then F cannot be fully propagated. Thus the constraint on the pending action $p()$ function is true.

Maximal confluence is a global property which is generally hard to establish. However, once established, the property is *preserved* by system evolutions:

Lemma 8. *1) F is an MCP of G implies F/s_F is an MCP of $G/f(s_F)$ for all $\hat{s}_F \xrightarrow{a}_F s_F$; 2) $\hat{s}_G \xrightarrow{a}_G s_G$ and F' is an MCP of G/s_G implies there exists an (not necessarily unique) MCP F of G s.t. $\hat{s}_F \xrightarrow{a}_F s_F, s_G = f(s_F)$ and $F/s_F = F'$.*

Proof. 1) Obviously F/s is a confluent process of $G/f(s)$. Assume F/s is not maximal for $G/f(s)$. Then there exists a MCP F' of $G/f(s)$ (via homomorphism f') such that F/s is a strict subgraph of F' . Assuming $S_{F'} \setminus S_{F/s}$ and $S_F \setminus S_{F/s}$ disjoint, we can merge F' and F by defining: $F'' = (S_F \cup S_{F'}, \Sigma, \Delta_F \cup \Delta_{F'}, \hat{s}_F)$. Since $S_{F''}$ is partitioned by $S_{F'}$ and $S_F \setminus S_{F/s}$ and no state of $S_{F'}$ can transit to a state of $S_F \setminus S_{F/s}$ whilst a state of $S_F \setminus S_{F/s}$ can only transit to a state of S_F , we can conclude that F'' is acyclic and $eb_{F''}(s) = eb_{F'}(s)$ for all $s \in S_{F'}$ and $eb_{F''}(s) = eb_F(s)$ for all $s \in S_F \setminus S_{F/s}$. Thus F'' is a confluent process of G via homomorphism $f \cup f'$, and F , as a strict subgraph of F'' , cannot be a MCP of G .

2) F' can be extended into a confluent process F^a of G by adding \hat{s}_{F^a} to $S_{F'}$, $(\hat{s}_{F^a}, a, \hat{s}_{F'})$ to $\Delta_{F'}$ and $(\hat{s}_{F^a}, \hat{s}_G)$ to f . Thus there exists a MCP F of G to which F^a is a subgraph. Obviously F' is a subgraph of $F/\hat{s}_{F'}$. And due to the maximality of F' and $F/\hat{s}_{F'}$, we have $F/\hat{s}_{F'} = F'$.

Now we can develop the notion of *maximal back-propagation* that will form the basis of our unfolding procedure in the next section.

- If F is a confluent process of G/s , then we say there is a *concurrent run* F from s , denoted $s \xRightarrow{F}_G$. If F is finite and $f(\check{s}_F) = s' \in S_G$, we further say that there is a *concurrent run* F from s to s' , denoted $s \xRightarrow{F}_G s'$.

- We say an action $a \in \Sigma$ is *fired* in a concurrent run $s \xrightarrow{F} G$ if an a -labelled transition is reachable in F . We say an action $a \in \Sigma$ is *blocked* in a concurrent run $s \xrightarrow{F} G$ if there exists a path $\hat{s}_F \xrightarrow{a_1 \cdots a_{n-1}}_F s_F \xrightarrow{a_n}_F s'_F$ in F s.t. $a_1, \dots, a_n \in \Sigma \setminus \{a\}$ and $a \diamond_{f(s_F)} a_n$ does not hold in G ; otherwise we say a is *unblocked* in $s \xrightarrow{F} G$.
- Given a confluent process F of G , we say an action $a \in \Sigma \setminus eb_F(s_F)$ is *postponed* at s_F (or, more accurately, the potential granular event with label a and origin s_F has postponed occurrence in F) if a is unblocked in F/s_F and there exists a granular event T' in F s.t. $lb(T') = a$ and $o(T') \sqsupset s_F$.
- Furthermore, we say $a \in \Sigma \setminus eb_F(s_F)$ is *p-pending* (partially pending) at $s_F \in S_F$ if a is unblocked but fired in $F/s_F(a)$ and there is no granular event T' in F s.t. $lb(T') = a$ and $o(T') \sqsupset s_F$, and we say $a \in \Sigma \setminus eb_F(s_F)$ is *pending* at $s_F \in S_F$ if a is neither fired nor blocked in $F/s_F(a)$.

For instance, given G (the leftmost graph) and its three confluent processes in the figure below, we can see c is pending at $s1$, p-pending at $s2$ and postponed at $s1'$ in the three confluent processes resp.

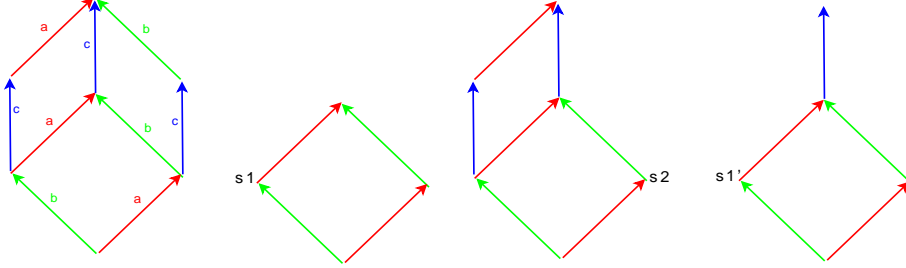


Fig. 3. Pending, p-pending and postponed

- A confluent process F is called *primary confluent process* if no action is postponed at any state in F .
- Given a finite confluent process F and an action $a \in eb_G(f(\check{s}_F))$, we define the *pending back-propagation* of a over F to be $bp_{pn}(a, F) = \{s_F \in S_F \mid a \text{ is pending at } s_F\}$, and the *maximal pending back-propagation* of a over F to be $mbp_{pn}(a, F) = \min_F(bp_{pn}(a, F))$.
- Similarly, given an action $a \in \Sigma$ which is p-pending at some state of F , we define the *p-pending back-propagation* of a over F to be $bp_{pp}(a, F) = \{s_F \in S_F \mid a \text{ is p-pending at } s_F\}$, and the *maximal p-pending back-propagation* of a over F to be $mbp_{pp}(a, F) = \min_F(bp_{pp}(a, F))$.

Lemma 9. *A confluent process F is an MRP iff there is no postponed or p-pending action at any $s_F \in S_F$.*

Proof. Assume F' is an MCP s.t. F is a subgraph of F' . Define $p(s_F) = eb_{F'}(s_F) \setminus eb_F(s_F)$ for all $s_F \in S_F$. If $p(s) \not\subseteq p(s')$ for some $s \xrightarrow{a}_F s'$, then F cannot be fully propagated. Thus the constraint on the pending action $p()$ function is true.

Lemma 10. *Given an action $a \in \Sigma$ postponed or p -pending at a state s_F of F , there exists a unique minimal relaxation F' of F , denoted $F' = F \uparrow_{s_F}^a$, s.t. $a \in eb_{F'}(s_F)$.*

Proof. The key idea of confluence is that, once an action is enabled, in subsequent evolution of the system the action will continue to be enabled until it is eventually executed. If $f(s_F) \xrightarrow{F/s_F(a)} G$, then it implies we can extend F by enabling a at s_F (and all its subsequent states in F): all the new states and transitions in the resultant confluent process can be unwinded from the original G system.

Lemma 11. *Given an action $a \in eb_G(f(\check{s}_F))$ and a state $s_F \in bp_{pn}(a, F)$, there exists a unique minimal elongation F' of F , denoted $F \xrightarrow{a}_{s_F} F'$, s.t. $a \in eb_{F'}(s_F)$.*

Proof. Let $B = eb_F(f(\check{s}_{F_0}))$ and F_B be a MCP prefix of F s.t. F_B/\check{s}_{F_0} is an B diamond. Find the set T of state and action pairs s.t. $(s, a) \in T$ iff s is a minimal state in S_{F_0} with $f(s) \xrightarrow{F_B/s(a)} G$. Let $D = \bigcup_{s \in \text{dom}(T)} \{T(s)\}$ and F_D be a MCP prefix of F s.t. F_D/\check{s}_{F_0} is an D diamond. Then F_D is a primary MCP prefix.

Theorem 1. *Given a finite primary confluent process F , if $s_F \in mbp_{pp}(a, F) \wedge F' = F \uparrow_{s_F}^a$ or $s_F \in mbp_{pn}(a, F) \wedge F \xrightarrow{a}_{s_F} F'$, then F' is a primary confluent process.*

4 Coalescing confluent processes

A confluent process records one possible history of system evolution. To see other possible evolutions and pinpoint where different evolutions come to deviate and split from each other, we need to coalesce a set of confluent processes into a branching structure. Coalescing operation merges the shared part of evolution histories, and in so doing, makes the ‘branching points’ explicit.

- Given a set \mathcal{F} of confluent processes of G , we use $pr(\mathcal{F})$ to denote the set of finite prefix of \mathcal{F} . Then we can construct a *general transition system* G'^3 , called the *coalescing* of \mathcal{F} , s.t. $S_{G'} = pr(\mathcal{F})$, $\hat{s}_{G'} = \hat{G}$, and $F \xrightarrow{a}_{G'} F'$ iff $F \preceq_E F'$ and $\check{s}_F \xrightarrow{a}_{F'} \check{s}_{F'}$. It is crucial to note that, for all $F \in S_{G'}$, F/G' is isomorphic to F and, therefore, a confluent process of G .
- The notions of granular events can be extended onto G' : $t_1 = F_1 \xrightarrow{a}_{G'} F'_1$ and $t_2 = F_2 \xrightarrow{a}_{G'} F'_2$ belong to a same granular event in G' iff t_1 belongs to T_1 in F'_1/G' , t_2 belongs to T_2 in F'_2/G' and $\min(T_1) = \min(T_2)$.

Although we can coalesce arbitrary sets of confluent processes, it makes more sense to coalesce a set of confluent processes that are 1) mutually incomparable w.r.t. \preceq and 2) able to *fully cover* the set of system evolutions. The second

³ The reason general transition systems are needed here is largely due to indeterminate evolution covers (introduced below).

requirement can be formalised in the same spirit as for the definition of unfolding. A confluent process F covers a set of system executions, i.e. those which are a linearisation of some prefix of F , denoted $lin(pr(\{F\}))$. \mathcal{F} fully covers the set of system evolutions if $\mathcal{L}(G) = lin(pr(\mathcal{F}))$. We call such set of confluent processes an *evolution cover* of G .

- An evolution cover \mathcal{F} of G is an MCP evolution cover if all $F \in \mathcal{F}$ are MCPs of G .
- A transition system G' is a *CP unfolding* of G if there exists an evolution cover \mathcal{F} of G s.t. G' is the coalescing of \mathcal{F} .
- A transition system G' is a *MCP unfolding* of G if there exists an MCP evolution cover \mathcal{F} of G s.t. G' is the coalescing of \mathcal{F} .
- If, furthermore, for all system executions L of G , there exists a unique $F \in pr(\mathcal{F})$ s.t. L is a linearisation of F , we call such evolution cover of G *determinate*.

For determinate evolution covers, we can give a simplified (alternative) definition to CP unfolding:

- We say an acyclic TS G is a *confluent tree* if, for all $s \in S_G$, s/G is confluent (denoting a concurrent run).
- We say a confluent tree G' is a *confluent tree unfolding* of TS G if G' is an unfolding of G .

Lemma 12. G' is a confluent tree unfolding of G iff there is a determinate evolution cover \mathcal{F} s.t. G' is the coalescing of \mathcal{F} .

The notion of *events* on top of confluent tree unfoldings is exactly the same as that on top of confluent processes, i.e. granular events quotiented by an equivalence which is the transitive closure of or-causal coupling relation.

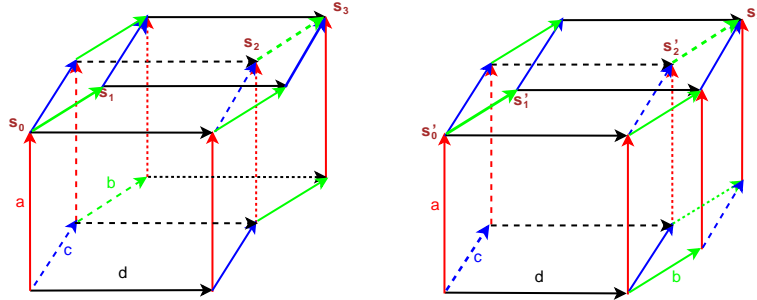


Fig. 4. Label ambiguity v.s. event ambiguity

However, the above definition is not extendable to general CP unfolding because of indeterminate evolution cover. For instance, when the two confluent

processes in the above figure are coalesced into G' , transitions $s_0 \xrightarrow{b} s_1$ and $s'_0 \xrightarrow{b} s'_1$ will collapse in G' into one transition, and so are states s_2 and s'_2 into one state, say $g_2 \in S_{G'}$. (Note that states s_3 and s'_3 will not collapse in G' ; they are mapped resp. to say $g_3, g'_3 \in S_{G'}$.) Since $s_0 \xrightarrow{b} s_1$ and $s_2 \xrightarrow{b} s_3$ belong to a same granular event in the left confluent process and $s'_0 \xrightarrow{b} s'_1$ and $s'_2 \xrightarrow{b} s'_3$ to a same in the right, using the above definition we can see that $s_2 \xrightarrow{b} s_3$ and $s'_2 \xrightarrow{b} s'_3$ belong to the same event in G' . Thus, from state g_2 by firing a same event we can reach two different states, i.e. g_3 and g'_3 . This is contradictory with the intuition of events. In summary, confluent tree unfolding supports the notion of events, whereas CP unfolding only supports the notion of granular events.

So far our problem statement and foundation work are developed mostly within the interleaving framework. But we have witnessed the usefulness of ‘event intuition’ in understanding notions like prefix, postpone and back-propagation for confluent processes. As we start to deal with more sophisticated *CP branching processes*, however, we will see that it is crucial (due to simplicity and intuitiveness) to reason directly in terms of events, concurrency, causality and conflict rather than in terms of transitions, commutativity, enabling and disabling.

Thus, we will move gradually into event-based models, e.g. configuration structures and granular configuration structure. Configuration structures is the non-interleaving incarnation of confluent tree unfolding and is thus built from events; while granular configuration structure is the non-interleaving incarnation of CP unfolding and is built from granular events.

Below we start with a quick introduction to the two structures, focusing on the correspondence with their transition system incarnations. Granular configuration structure will be the basis of our MCP unfolding (that requires indeterminate evolution covers, c.f. the example in Figure 1). We will present a formal and detailed introduction of granular configuration structure in the next section⁴.

A configuration structure (E, C) consists of a set E of events and a set C of configurations. Each configuration $c \in C$ is a subset of events, which represents the global state reached after firing exactly the set c of events. The empty configuration represents the initial state.

For example, the left graph in the Figure below is a confluent tree G . We can coarsely partition transitions in G into events as shown in the middle graph, or we can finely group them into granular events, which do not form a partitioning of transitions (e.g. $e3$ and $e3'$ share a transition), as shown in the right graph. In the middle graph each state in G is mapped to a configuration. Given a state s mapped to c , if s can transit to s' via a transition belonging to e , then the s' is mapped to $c \cup \{e\}$. The soundness of this rule is implied by the fact that, no matter what system execution one uses to reach a given state in F , the set of events fired by the execution is the same.

Similarly in the right graph each state s in G is mapped to a granular configuration c . But the property that all system executions to a same state fire the same set of events is no longer true. c here denotes, instead, the set of granular

⁴ A formal and detailed introduction of configuration structures can be found in [10].

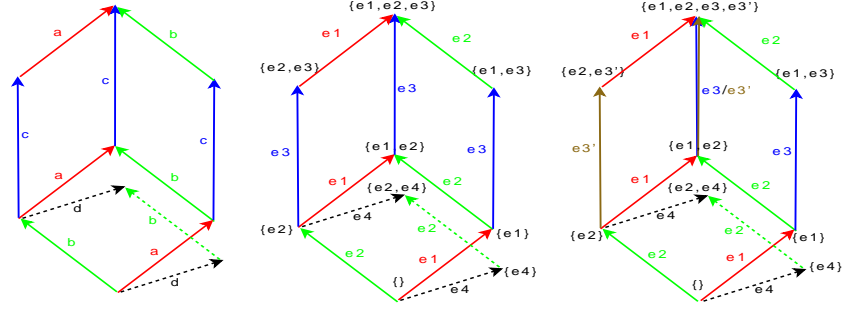


Fig. 5. From transitions to events and granular events

events whose member (i.e. transition) has occurred in s/G . Thus it is possible that, by firing one transition in G , we can fire more than one event in the granular configuration structure, e.g. from $\{e1, e2\}$ to $\{e1, e2, e3, e3'\}$ by c and from $\{e1, e3\}$ to $\{e1, e2, e3, e3'\}$ by b in the right graph.

Discussion: The notion of confluent tree unfolding can be of independent interests. Indeed it provides a powerful tool for analysing previous works such as [21, 14] and [10].

In [21, 14] the class of confluent processes one can produce by unfolding a TS G is highly constrained. It is because an independence relation is imposed on top of G , i.e. the so called *transition systems with independence* (TSI). The independence relation marks (statically) a selected subset of diamonds in G as ‘true diamonds’, and requires all diamonds in confluent processes originating from true diamonds. Furthermore, Axiom 3 of TSI requires that no true diamond can be unfolded sequentially, i.e. if two consecutive edges of the diamond is unfolded in F , then the whole diamond is unfolded in F .

The work in [10] removes the static independence restriction on transitions. Thus a confluent process can utilise any possible diamond in G , and a diamond can be unfolded sequentially in one confluent process while concurrently in another one (c.f. the example in Figure 10 of [10]).

Furthermore, Axiom 4 of TSI imposes a transitivity-like condition on the set of true diamonds so that they form a global network of diamonds and the existence of a true diamond at one location implies the existence of a set of true diamonds at its neighboring locations. Therefore, Axiom 4 combined with Axiom 3 ensures that 1) or-causality does not occur in confluent processes (and thus granular events coincide with events), and 2) non-local conditions become reducible to local ones (since the non-local part is guaranteed by the transitivity). One example is, given a confluent process F of a TSI G and an event T in F , T is postponed at a state $s \in S_F$ that is adjacent to $o(T)$ via transition $s \xrightarrow{a} o(T)$ iff $a \diamond_{f(s)} lb(T)$ in G .

5 Unfolding Procedure

In this section we first introduce *granular configuration structures* which is an adaptation of labelled configuration structures [18, 17, 10]. Granular configuration structures 1) restore the causality relation on events which can greatly simplify the definition of advanced notions like prefixes, immediate conflicts, etc. and 2) improve the expressiveness so that CP/MCP unfoldings can be fully captured. Then we give the *MCP unfolding* procedure to unfold transition systems into granular configuration structures.

5.1 Granular configuration structures

Definition 2. A granular configuration structure (or simply GCS) over alphabet Σ is a triple (E^{\leq}, C, lb) , where

- E is a partially ordered set of granular events (or henceforth simply events), where \leq is the well-founded causality relation,
- lb is a labelling function mapping events of E into labels of Σ ,
- and C is a set of granular configurations (or simply configurations), where each configuration $c \in C$ is a finite \leq -downward closed subset of E and $e \in E$ implies $[e] \in C$.

A configuration c can be thought of as representing a state reached after the execution of exactly the set c of granular events. The empty configuration $\{\}$ represents the initial state and is a required member of C in our model.

Below we fix a GCS, $cs = (E^{\leq}, C, lb)$, and introduce some basic notions for GCSes.

- We say cs is *finitely branching* if the Hasse diagram of E^{\leq} is finitely branching. In such a GCS, concurrency and conflict are bounded and *infinite configurations* are derivable from finite ones.
- $[e]$ denotes the \leq -downward closure of $\{e\}$ while $[e]^-$ denotes $[e] \setminus \{e\}$.
- Given $X \subseteq E$, $[X]$ and $[X]^-$ denote $\bigcup_{e \in X} [e]$ and $\bigcup_{e \in X} [e]^-$ resp.
- We say a nonempty finite subset $\delta \subseteq E$ is a *consistent set* if there is some $c \in C$ such that $\delta \subseteq c$. Otherwise, δ is an *inconsistent set*.
- A consistent \leq -downward closed $\delta \subseteq E$ is called a *pre-configuration*.
- We say an event e is *activated* at pre-configuration δ , or $e \in ac(\delta)$, if $[e]^- \subseteq \delta$ and $c \uplus \{e\}$ is consistent.
- We say cs is *well-activated* if $lb(e) = lb(e') \wedge e \parallel_{\leq} e' \wedge \{e, e'\}$ is consistent implies $[e]^- \parallel_{\subseteq} [e']^-$.
- We say an activated event e at pre-configuration δ is *pending* at δ , or $e \in ac_{pn}(\delta)$, if $\forall e' \in \delta : e \parallel_{\leq} e' \implies lb(e) \neq lb(e')$.
- We say an activated event e is *p-pending* at pre-configuration δ , or $e \in ac_{pp}(\delta)$, if $\exists e' \in \delta : e \parallel_{\leq} e' \wedge lb(e) = lb(e')$. Given a pre-configuration δ , we say δ is *p-pending closed* if $ac_{pp}(\delta) = \{\}$; otherwise we use δ^* to denote the *p-pending closure* of δ .

- We say there is a *transition* from c to c' , written $c \xrightarrow{a}_C c'$, if $c \subseteq c'$ and there exists an event $e \in ac_{pn}(c)$ s.t. $lb(e) = a$ and $\{e\} \subseteq c' \setminus c \subseteq (c \uplus \{e\})^* \setminus c^*$.

The definition of transitions here is unconventional, esp. in comparison with configuration structures. A transition from c to c' may involve multiple events ($c' \setminus c$). Some of them, those pending events from $ac_{pn}(c)$, are the ‘driving events’ of the transition while others, those p-pending events from $(c \uplus \{e\})^* \setminus (c^* \cup \{e\})$, are ‘auxiliary ones’ piggybacked on the transition. Note that only those freshed generated p-pending events (due to the driving ones) can be piggybacked, not old one from $c^* \setminus c$.

Thus a GCS gives rise to an acyclic transition system, and the definitions like ‘subsequent to’ relation \sqsubseteq , (system) execution, etc. carry over. Furthermore, note that $c \subseteq c'$ does not implies there exists an execution from c to c' in GCSes; this is very different from configuration structures.

- We write $eb(c) = \{a \in \Sigma \mid c \xrightarrow{a}_C\}$ to denote the set of enabled actions at c . $succ(c) = \{c' \in C \mid c \xrightarrow{a}_C c'\}$ denotes its set of successor configurations.
- We say cs is *well-connected* if all the configurations in C are \rightarrow_C -reachable from $\{\}$ and, for all $c \in C$ and $e \in ac_{pn}(c)$, there exists $c' \in C$ s.t. $c \xrightarrow{lb(e)}_C c' \wedge e \in c'$.

Based on the above, we can say cs is *well-formed* if it is finitely branching, well-activated and well-connected. Well-formed GCSes have roughly the same expressiveness as (general) event structures from [20]. We prefer to use GCSes in this paper mainly because of its affinity to transition systems. We give a few basic properties of well-formed GCSes, esp. those in comparison with configuration structures.

- Given a finite $D \subseteq C$, we say D is *downward-closed* (w.r.t. \sqsubseteq) if $c \sqsubseteq c' \in D \implies c \in D$. We use D^\downarrow to denote the \sqsubseteq -downward closure of D .
- We say a finite subset of configurations $D \subseteq C$ are *compatible* if $\bigcup D$ is consistent and disjoint from $\bigcup_{c \in D} (c^* \setminus c)$.
- We say cs is *closed under bounded union* if, for all compatible subsets $D \subseteq C$, $\exists c' \in C : \bigcup D \subseteq c' \subseteq (\bigcup D)^* \setminus \bigcup_{c \in D} (c^* \setminus c)$.
- We say cs is *free of auto-concurrency* if $lb(e) = lb(e')$ and $e \parallel_{\leq} e'$ implies $\nexists c \in C : [e']^- \cup [e] \subseteq c \wedge e' \in ac(c)$.⁵

Lemma 13. *cs is free of auto-concurrency and closed under bounded union.*

Proof. Due to well-formedness of GCSes.

Lemma 14. *If there is a non-empty execution from c to c' such that $e \in ac(c)$ and $c' \cup \{e\}$ is consistent, then we have either $e \in c'$ or $e \in ac(c')$.*

⁵ GCSes with auto-concurrency can be useful, on the other hand, for unfolding systems like general Petri Net.

Proof. If c_i and c_{i+1} are two adjacent configurations with $e \notin c' \setminus c$ on the path and e is enabled at c_i (i.e. $(c_i \cup \{e\})^* \in C$), closure under bounded union gives us that e is enabled at c_{i+1} .

GCSes are the non-interleaving incarnations of the coalescing of confluent processes: each configuration in a GCS uniquely corresponds to an acyclic confluent transition system.

Lemma 15. *Given any $c \in C$, $cs \triangleright \{c\}^\downarrow$, i.e. the restriction of cs to $\{c\}^\downarrow$, gives rise to an acyclic confluent transition system, i.e. $CP(c) = (\{c\}^\downarrow, \Sigma, \rightarrow_{\{c\}^\downarrow}, \{\})$, where $CP()$ is an injective function.*

Proof. Use bounded union closure and free of auto-concurrency.

For the rest of this paper we only consider well-formed GCSes and simply call them GCSes. Advanced notions of GCSes can be easily defined by using the restored causality relation:

- Given a finite \leq -downward closed subset $X \subseteq E$, we say X is *p-pending event closed* (or simply pp-event closed) if, for all pre-configuration $\delta \subseteq X$ and event $e \in X$, $e \in ac_{pn}(\delta)$ implies $(\delta \cup \{e\})^* \subseteq X$. We denote the *pp-event closure* of X as X^* .
- A finite subset $X \subseteq E$ is a *prefix* if X is both \leq -downward closed and pp-event closed.
- A finite \leq -antichain $K \subseteq E$ is an *immediate conflict* (IC) if $[K]^*$ is a minimal prefix that is not conflict-free.

Lemma 16. *K is an IC implies $K \subseteq ac_{pn}([K]^-)$.*

Based on these notions, we can recover a purely event-based definition of GCS-like structures (i.e. without resorting to the use of configurations):

Granular Event Structures (GESes): A granular configuration structure (say cs) can be re-formulated (say using transformation $\mathcal{CE}(cs)$) into a *granular event structure*: a granular event structure is a triple, $es = (E^\leq, IC, lb)$, where IC is a set of immediate conflicts (IC). An immediate conflict $K \in IC$ is a finite \leq -antichain of events satisfying that, for all $K \in IC$, $[K]$ contains no IC other than K and, for all $e \in E$, $[e]$ contains no IC.

Conversely, we can also recover a granular configuration structure from es (say using transformation $\mathcal{EC}(es)$). Given es , we say a finite subset $X \subseteq E$ is *consistent* if $[X]$ contains no IC. This enables us to recover the definition of *pre-configuration*, *activated/pending/p-pending* events and *well-activatedness*. On well-activated es , we say a pre-configuration $\delta \subseteq E$ is a *configuration* if there exists another pre-configuration $\delta' \supseteq \delta$ s.t. $(ac(\delta') \cup \delta') \cap ac_{pp}(\delta) = \{\}$. Finally we say es is *well-formed* if it is well-activated, finite-branching and satisfying $e \in E \implies [e]$ is a configuration.

We can show that well-formed granular event structures correspond exactly to well-formed granular configuration structures:

Theorem 2. *$cs = \mathcal{EC}(\mathcal{CE}(cs))$ and $es = \mathcal{CE}(\mathcal{EC}(es))$.*

5.2 Unfolding TSes into GCSes

The aim of our procedure is to construct the Hasse diagram of configurations in a roughly bottom up fashion. Starting from the empty configuration, we move up step by step, deriving larger configurations from smaller ones. Each configuration generated corresponds to a finite MCP prefix (up to isomorphism).

However, since transitions between configurations follow ‘big-step semantics’ (i.e. firing multiple events), a simpler and more elegant approach is to first build the Hasse diagram of pre-configurations, where the transitions follow ‘small-step semantics’. Each pre-configuration correspond to a finite PCP prefix. Then, we remove all pre-configurations that are not configurations (called *nonstable pre-configurations*) and re-connect what are left, i.e. configurations, by big steps.

The search for new PCP prefixes is guided by a key sub-procedure of extending one PCP (say F) into another PCP (say F'). Firstly, we calculate the maximal back-propagation of actions for F and, if there is no corresponding events existing for those points, create new ones accordingly. Then we extend F by those events to generate F' (i.e. elongate or relax F depending on the events being pending or p-pending), which is a PCP according to Theorem 1. Note that in generating F' there is a ‘prefix-closure’ effect. That is, F' might have more than one immediate prefix and some of them (other than F) might be non-PCP and, therefore, not generated yet. Thus in generating F' we also need to generate some of its prefixes.

Now let us formalise the unfolding procedure and define the notions of *MCP unfolding* and *MCP branching processes*:

- Given a TS G , a *labelled GCS* over G is a tuple $lcs = (E^{\leq}, C, lb, st)$, where (E^{\leq}, C, lb) constitutes a GCS over Σ and $st : C \rightarrow S_G$ is a function mapping configurations to states.
- Given lcs over G , we say lcs is an *MCP unfolding* of G if $(C, \Sigma, \rightarrow_C, \{\})$ is an MCP unfolding of G via homomorphism $f = st$.
- Given lcs over G , we say $lcs = (E^{\leq}, C, lb, st)$ is an *MCP branching process* of G if there exists an *MCP unfolding* $lcs' = (E'^{\leq}, C', lb', st')$ of G s.t. E is a prefix of (E'^{\leq}, C', lb') , $C = C' \triangleright E$, $lb = lb' \triangleright E$ and $st = st' \triangleright C$.

Our procedure, $lcs = \mathcal{U}(G)$ (c.f. Figure 6), unfolds TSes in a maximally concurrent fashion into a labelled GCS over G .

The basic data structure is $(E, preC, lb, st)$. E and $preC$ store resp. the set of generated events and the set of generated pre-configurations. The back-propagation information for each pre-configuration is stored in function bp_{pn} and bp_{pp} resp. We create new events based on such information, which are then passed on from smaller pre-configurations to larger ones and stored in function ac_{pn} and ac_{pp} resp. as activated events.

Then, adding activated events to existing pre-configurations produces the set of potential *extensions*, EXT. Some of the extensions are PCPs, i.e. those in the set PCP. The definition of PCP is recursive, utilising Theorem 1 and starting from the empty pre-configuration. But, due to the ‘prefix closure’ effect mentioned above, PCP extensions cannot be *realised* ‘eagerly’ by immediately throwing

them into $preC$. Rather, the realisation proceeds in steps by first realising the prefixes (i.e. sub-configuration⁶) of the PCP extensions, so that the expansion of $preC$ will preserve the \preceq -downward closedness. We say a PCP prefix is *ready for realisation* if it is unrealised but all its sub-configuration are realised. The set of ready-for-realisation PCP prefixes is given in NXT.

Thus, starting from an empty $preC$ we pick one pre-configuration (say c') a time from NXT and realise by adding c' to $preC$ (line 3-5 of function UNFOLD in Figure 6). In the mean time we calculate (c.f. line 8-9 of function REALISE) the set of activated events inheritable by c' from its immediate sub-configurations (i.e. $\bullet c'$), and also derive (line 4-7 of function REALISE) the new bp_{pn} and bp_{pp} functions based on those of $\bullet c'$. Some useful notations used in such calculation/derivation is given below.

Given any $c, c' = c \uplus \{e\} \in preC$, we define

- $prp_{pn}(c', c, a) =$
 - $\{\{e\}\}$ if $e \in ac_{pn}(c) \wedge \neg lb(e) \diamond_{st(c)} a \vee e \in ac_{pp}(c) \wedge mbp_{pn}(c, a) = \{c\}$, or
 - $min(mb_{pn}(c, a) \cup \{\{e\}\})$ if otherwise;
- $prp_{pp}(c', c, a) =$
 - $bp_{pp}(c, a)$ if $lb(e) \neq a$,
 - $\{c_0 \in bp_{pp}(c, a) \mid c_0 \parallel_{\sqsubseteq} [e]^{-}\}$ if $e \in ac_{pp}(c) \wedge lb(e) = a$, or
 - $\{c_0 \in bp_{pn}(c, a) \cup bp_{pp}(c, a) \mid c_0 \parallel_{\sqsubseteq} [e]^{-}\}$ if $e \in ac_{pn}(c) \wedge lb(e) = a$;
- $hrt_{pn}(c', c) = \{e' \in ac_{pn}(c) \mid e \in ac_{pp}(c) \vee (e \in ac_{pn}(c) \wedge lb(e) \diamond_{st(c)} lb(e'))\}$
- $hrt_{pp}(c', c) = \{e' \in (ac_{pn}(c) \cup ac_{pp}(c)) \mid (e' \in ac_{pn}(c) \implies e \in ac_{pn}(c) \wedge lb(e) = lb(e')) \wedge (lb(e) = lb(e') \implies [e']^{-} \parallel_{\sqsubseteq} [e]^{-})\}$.

Note that $hrt_{pn}(c', c)$ produces the set of pending events c' can inherit from c while $hrt_{pp}(c', c)$ produces the set of p-pending events c' can inherit from c . On the other hand, $prp_{pn}(c', c, a)$ produce the information about how action a can be back-propagated through edge (c, c') into the sub-configurations while $prp_{pp}(c', c, a)$ about the p-pending points of a through edge (c, c') .

Then, if c' is a PCP and has maximal back-propagation not covered by activated events inherited from $\bullet c'$, we create new ones to cover them (c.f. procedure GENEVENT). An event (say e) can be created only if its origin (say c) is a *configuration* according to c' : the condition is implemented as the predicate $c \text{ ISCFGIN } c'$. After e is created at c , we propagate e upward to its super-configurations (c.f. line 4-8 of procedure GENEVENT).

Finally, after the set of pre-configurations fully generated, we filter out non-stable pre-configurations and produce the set of configurations (c.f. line 6 of function UNFOLD). The intuition is that a PCP pre-configuration is a MRP if $ac_{pp}(c') = \{\}$ and the prefixes of MRP pre-configurations are configurations.

We can illustrate the procedure by unfolding the broken cube originally from Figure 1. In the step 0 of Figure 7 the initial state is mapped to configuration $\{\}$. The set of activated events at $\{\}$, i.e. $ac_{pn}(\{\})$ and $ac_{pp}(\{\})$, is initialised

⁶ Actually it should be sub-pre-configurations, i.e. pre-configurations which are subset of the original pre-configurations.

Data Structure:

$$(E, preC, lb, st) = (\{\}, \{\{\}\}, \{\}, \{\{\{\}, \hat{s}\}\});$$

$$ac_{pn} = \{\}; ac_{pp} = \{\}; bp_{pn} = \{\}; bp_{pp} = \{\};$$

Derived Values, Functions and Predicates:

$$EXT = \{c \cup \{e\} \mid c \in preC \wedge e \in ac_{pn}(c) \cup ac_{pp}(c)\}$$

$$PCP = \{c \cup \{e\} \mid c \in PCP \wedge (e \in ac_{pn}(c) \wedge [e]^- \in mbp_{pn}(c, lb(e))$$

$$\quad \vee e \in ac_{pp}(c) \wedge [e]^- \in mbp_{pp}(c, lb(e)))\}$$

$$NXT = \{c' \in EXT \setminus preC \mid \exists c'' \in PCP : c' \subseteq c'' \wedge \forall c \in EXT : c \subset c' \implies c \in preC\}$$

$$c \text{ ISCFGIN } c' = \forall e \in c' \setminus c : [e]^- \notin bp_{pp}(c, lb(e)) \wedge \forall a \in \Sigma : bp_{pp}(c, a) \cap bp_{pp}(c', a) = \{\}$$

$$\bullet c' = \{c \mid c \in preC \wedge c \uplus \{e\} = c'\}$$

Function UNFOLD((S, Σ, Δ, ŝ))**Begin**

- 1 **Foreach** $a \in eb(\hat{s})$ **do** Set $bp_{pn}(\{a\}, a) = \{\{\}\}$;
- 2 **GENEVENT**($\{\}$);
- 3 **While** $NXT \neq \{\}$
- 4 Pick any $c' \in NXT$ and **REALISE**(c');
- 5 **If** $c' \in PCP$ **Then** **GENEVENT**(c');
- 6 Set $C = \{c \in preC \mid \exists c' \in PCP : c \subseteq c' \wedge ac_{pp}(c) \cap c' = \{\} = ac_{pp}(c')\}$;
- 7 **Return** (E, C, lb, st)

End**Procedure REALISE(c')****Begin**

- 1 **Assume** $c' = c \uplus \{e\}$ for some $c \in preC$;
- 2 **If** $e \in ac_{pn}(c)$ **Then** $s' = \Delta(st(c), lb(e))$ **Else** $s' = st(c)$;
- 3 Add c' to $preC$ and (c', s') to $st()$;
- 4 **Foreach** $a \in eb(st(c'))$ **do**
- 5 Set $bp_{pn}(c', a) = \bigcap_{c \in \bullet c'} \{c_1 : preC \mid \exists c_0 \in prp_{pn}(c', c, a) : c_0 \subseteq c_1 \subseteq c'\}$;
- 6 **Foreach** $a \in \Sigma$ s.t. $D = \bigcup_{c \in \bullet c'} prp_{pp}(c', c, a) \neq \{\}$ **do**
- 7 Set $bp_{pp}(c', a) = \{c_0 \in D \mid \forall c \in \bullet c' : c_0 \subseteq c \implies c_0 \in prp_{pp}(c', c, a)\}$;
- 8 Set $ac_{pn}(c') = \{e' \in \bigcup_{c \in \bullet c'} hrt_{pn}(c', c) \mid \forall c \in \bullet c' : [e']^- \subseteq c \implies e' \in hrt_{pn}(c', c)\}$;
- 9 Set $ac_{pp}(c') = \{e' \in \bigcup_{c \in \bullet c'} hrt_{pp}(c', c) \mid \forall c \in \bullet c' : [e']^- \subseteq c \implies e' \in hrt_{pp}(c', c)\}$

End**Procedure GENEVENT(c')****Begin**

- 1 **Foreach** $(a, c) \in mbp_{pp}(c') \cup mbp_{pn}(c')$
s.t. $\nexists e \in E : (lb(e), [e]^-) = (a, c) \wedge c \text{ ISCFGIN } c'$ **do**
- 2 Create a new event e , and add e to E and (e, a) to $lb()$;
- 3 Add e to $ac_{pn}(c)$ and set $D = \{c\}$;
- 4 **While** $M = \min(\{x \in preC \mid x \supset c\} \setminus D) \neq \{\}$ **do**
- 5 Pick any $c'' \in M$ and add c'' to D ;
- 6 **If** $\forall c \in \bullet c'' : [e]^- \subseteq c \implies e \in hrt_{pn}(c'', c)$ **Then** Add e to $ac_{pn}(c'')$
- 7 **Else If** $\forall c \in \bullet c'' : [e]^- \subseteq c \implies e \in hrt_{pp}(c'', c)$
- 8 **Then** Add e to $ac_{pp}(c'')$ **Else** Add $\{c \in preC \mid c \supseteq c''\}$ to D

End**Fig. 6.** An unfolding procedure

to be empty (i.e. no inheritance). Since the three enabled actions at the initial state are maximally back-propagated to $\{\}$ but there is no activated events at $\{\}$ to match them, we create three new events $e1$, $e2$ and $e3$ at $\{\}$ (note the use of symbol !) and added them to $ac_{pn}(\{\})$. Thus the initial state is labelled as $\{\}/\{e1, e2, e3\}$ (in the style of $c/ac_{pn}(c)$). Firing one of the generated events say $e1$ leads to a new extension, say $\{e1\}$, which is a new member of NXT .

In the step 1, we pick a member say $\{e1\}$ of NXT and realise it. $\{e1\}$ is mapped to s_1 and $ac_{pn}(\{e1\})$ inherits $\{e2, e3\}$ from $\{\}$, which fully covers the maximal back-propagation of the two actions enabled at s_1 . Thus, although $\{e1\}$ is a PCP and procedure `GENEVENT` is called, no new event will be created. Similarly we can also realise $\{e2\}$ and $\{e3\}$.

Now $\{e1, e3\}$ is a member of NXT , which we can pick in the step 2 to realise. Note that $\{e1, e3\}$ can inherit $e2$ from $\{e1\}$ but not from $\{e3\}$ (since $e2$ and $e1$ do not form a diamond at $\{e3\}$). This inconsistency leads to $e2$ not being added to $ac_{pn}(\{e1, e3\})$. $\{e1, e3\}$ is a PCP and in calling `GENEVENT`, however, a new event $e2'$ is created at $\{e1\}$ to cover the maximal back-propagation of the enabled b action at s_5 (mapped to $\{e1, e3\}$). $e2'$ can be inherited and added to $ac_{pn}(\{e1, e3\})$. Similarly, the back-propagation from $\{e1, e2\}$ and from $\{e2, e3\}$ leads to the creation of $e3'$ at $\{e1\}$ and $e1'$ at $\{e2, e3\}$ resp.

$e1'$ at $\{e2, e3\}$ can lead to a PCP extension but not so for $e2'$ and $e3'$ at $\{e1\}$. Instead, they are PCP prefixes since $\{e1, e2, e3'\}$ and $\{e1, e2', e3\}$ are PCP extensions. Therefore, all the possible extensions so far are inside NXT . In realising these extensions, $\{e1, e3'\}$ and $\{e1, e2'\}$ need to be realised before $\{e1, e2, e3'\}$ and $\{e1, e2', e3\}$ resp. to preserve the \preceq -downward closedness of $preC$.

In the step 3, obviously $\{e1, e2'\}$ inherits $e3$ and $e3'$ from $\{e1\}$; and $\{e1, e3'\}$ inherits $e2$ and $e2'$. But $e3'$ at $\{e1, e2'\}$ and $e2'$ at $\{e1, e3'\}$ will lead to extensions outside NXT . So the outcome is a GCS with three maximal configurations. Two of them, $\{e1, e3, e2'\}$ and $\{e1, e2, e3'\}$, are mapped to a same terminating state s_7 . (In contrast, state s_5 are split into $\{e1, e3\}$ and $\{e1, e3'\}$ while s_6 into $\{e1, e2\}$ and $\{e1, e2'\}$.)

The broken cube example does not have or-causality; thus the part of the procedure related to non-stable pre-configuration and p-pending points/events has not been utilised. We give a second example in Figure 8 to do so.

The original transition system is given as the top-left graph in the figure. The top-right graph is its MCP unfolding, which is the coalescing of two MCPs ($\{a1, b1, c1, c2, d2\}$ and $\{a1, b1, c2, d1, d2\}$). The MCP of $\{a1, b1, c1, c2, d2\}$ is drawn with thick-line edges and strong-colored configurations.

The bottom graph is the Hasse diagram of pre-configurations (with some edges missing). The faint-colored pre-configurations in bottom graph are non-stable pre-configurations; they will be removed in order to produce the Hasse diagram of configurations. Note further that $\{a1, b1, c2\}$ (in red-color) is a configuration in the MCP of $\{a1, b1, c1, c2, d2\}$ but not so in that of $\{a1, b1, c2, d1, d2\}$, even though it is a pre-configuration being used in the generation of both MCPs. Note that the set of pre-configurations being used to generate the MCP of $\{a1, b1, c1, c2, d2\}$ are those connected up by edges in the graph.

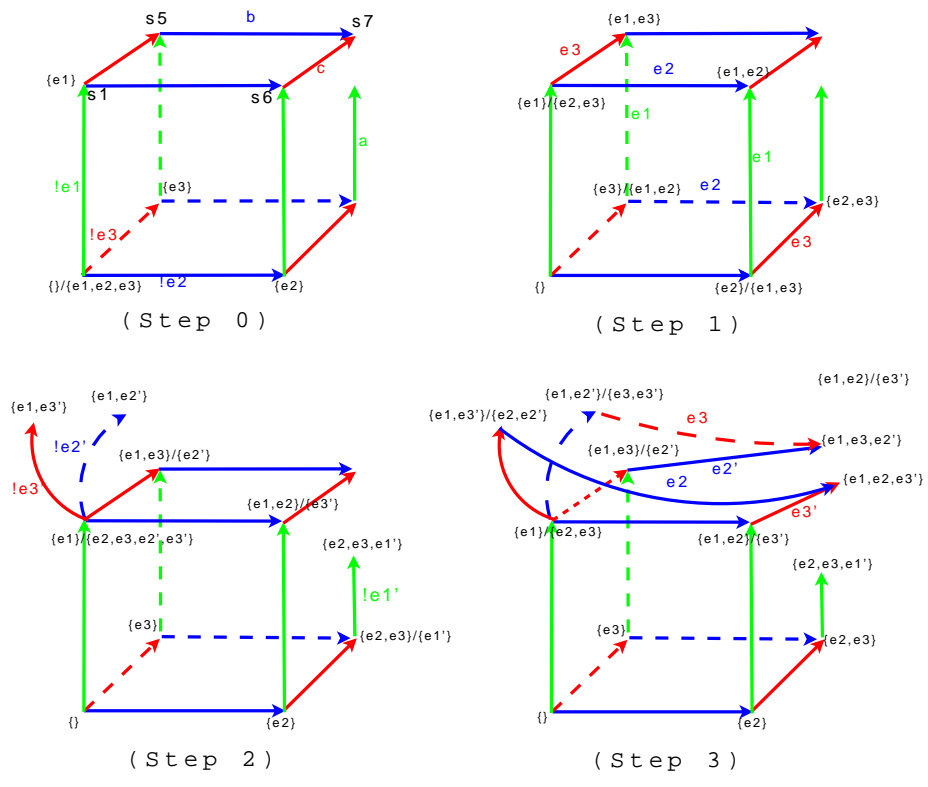


Fig. 7. Unfolding of the broken cube

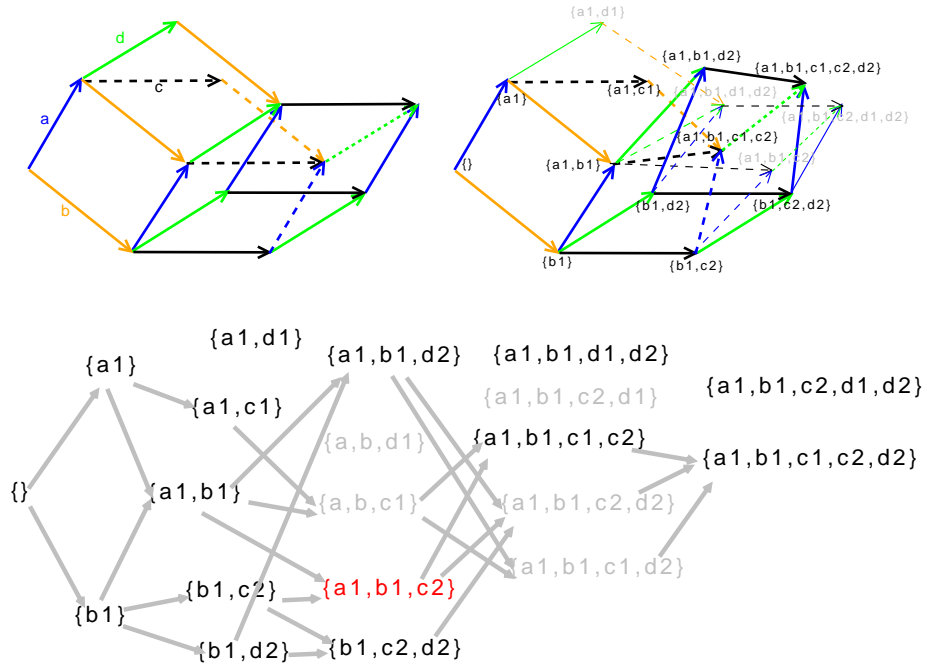


Fig. 8. P-pending event and nonstable pre-configuration

We can show that the output of function UNFOLD indeed is the MCP unfolding of G :

Theorem 3. $U(G)$ is an MCP unfolding of G .

References

1. Eric Badouel, Luca Bernardinello, and Philippe Darondeau. The synthesis problem for elementary net systems is np-complete. *Theor. Comput. Sci.*, 186(1-2):107–134, 1997.
2. Eric Badouel and Philippe Darondeau. Theory of regions. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 529–586. Springer, 1998.
3. Eike Best and Raymond R. Devillers. Sequential and concurrent behaviour in petri net theory. *Theor. Comput. Sci.*, 55(1):87–136, 1987.
4. Eike Best and César Fernández. *Nonsequential processes : a Petri net view*, volume 13 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1988.
5. Andrzej Ehrenfeucht and Grzegorz Rozenberg. Partial (set) 2-structures. part i: Basic notions and the representation problem. *Acta Inf.*, 27(4):315–342, 1989.
6. P. Godefroid. *Partial-order Methods for the Verification of Concurrent Systems: an Approach to the State-explosion Problem*, volume 1032 of *LNCS*. Springer, 1996.
7. Ursula Goltz and Wolfgang Reisig. The non-sequential behavior of petri nets. *Information and Control*, 57(2/3):125–147, 1983.

8. J.F. Groote and M. P. A. Sellink. Confluence for process verification. *Theoretical computer science*, 170(1-2):47–81, 1996.
9. Jeremy Gunawardena. Causal automata. *TCS*, 101(2):265–288, 1992.
10. Henri Hansen and Xu Wang. On the origin of events: branching cells as stubborn sets. In *Applications and Theory of Petri Nets, 32st International Conference (PETRI NETS 2011)*, Lecture Notes in Computer Science (to appear), 2011.
11. Xinxin Liu and David Walker. Partial confluence of processes and systems of objects. *TCS*, 206(1-2):127–162, 1998.
12. Robin Milner. *Concurrency and Communication*. Prentice-Hall, 1988.
13. D. A. Peled. All from one, one for all: on model checking using representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification*, volume 697 of *LNCS*, pages 409–423. Springer, 1993.
14. Vladimiro Sassone, Mogens Nielsen, and Glynn Winskel. Models for concurrency: Towards a classification. *Theor. Comput. Sci.*, 170(1-2):297–348, 1996.
15. A. Valmari. Stubborn sets for reduced state space generation. In *ICATPN*, volume 483 of *LNCS*, pages 491–515. Springer, 1989.
16. R. J. van Glabbeek. History preserving process graphs. available at <http://kilby.stanford.edu/~rvg/pub/history.draft.dvi>, 1996.
17. R. J. van Glabbeek and G. D. Plotkin. Configuration structures, event structures and petri nets. *Theoretical Computer Science*, 410(41):4111–4159, 2009.
18. Rob van Glabbeek and Ursula Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Informatica*, 37(4):229–327, 2001.
19. Rob J. van Glabbeek and Gordon D. Plotkin. Event structures for resolvable conflict. In *MFCS 2004*, volume 3153 of *Lecture Notes in Computer Science*, pages 550–561, 2004.
20. Glynn Winskel. Event structures. In *Advances in Petri Nets*, volume 255 of *LNCS*, pages 325–392. Springer, 1987.
21. Glynn Winskel and Mogens Nielsen. Models for concurrency. In *Handbook of logic in Computer Science*, volume 4. Clarendon Press, 1995.
22. A. Yakovlev, M. Kishinevsky, A. Kondratyev, L. Lavagno, and M. Pietkiewicz-Koutny. On the models for asynchronous circuit behaviour with or causality. *FMSD*, 9(3):189–233, 1996.