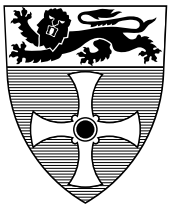
UNIVERSITY OF
NEWCASTLE



University of Newcastle upon Tyne

COMPUTING SCIENCE

Behaviour-Preserving Transition Insertions in Unfolding Prefixes

V. Khomenko

TECHNICAL REPORT SERIES

No. CS-TR-952 March, 2006

TECHNICAL REPORT SERIES

No. CS-TR-952 March, 2006

Behaviour-Preserving Transition Insertions in Unfolding Prefixes

Victor Khomenko

Abstract

Some design methods based on Petri nets modify the original specification by behaviour-preserving insertion of new transitions. If the unfolding prefix is used to analyse the net, it has to be re-unfolded after each modification, which is detrimental for the overall performance.

The approach presented in this paper applies the transformations directly to the unfolding prefix, thus avoiding re-unfolding. This also helps in visualisation, since the application of the transformation directly to the prefix changes it in a way that was `intuitively expected' by the user, while re-unfolding can dramatically change the shape of the prefix. Moreover, rigourous validity checks for several kinds of transition insertions are developed. These checks are performed on the original unfolding prefix, so one never has to backtrack due to the choice of a transformation which does not preserve the behaviour.

Printed and published by the University of Newcastle upon Tyne,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

KHOMENKO, V.

Behaviour-Preserving Transition Insertions in Unfolding Prefixes
[By] V. Khomenko

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2006.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-952)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-952

Abstract

Some design methods based on Petri nets modify the original specification by behaviour-preserving insertion of new transitions. If the unfolding prefix is used to analyse the net, it has to be re-unfolded after each modification, which is detrimental for the overall performance.

The approach presented in this paper applies the transformations directly to the unfolding prefix, thus avoiding re-unfolding. This also helps in visualisation, since the application of the transformation directly to the prefix changes it in a way that was `intuitively expected' by the user, while re-unfolding can dramatically change the shape of the prefix. Moreover, rigourous validity checks for several kinds of transition insertions are developed. These checks are performed on the original unfolding prefix, so one never has to backtrack due to the choice of a transformation which does not preserve the behaviour.

About the author

Obtained MSc with distinction in Computer Science, Applied Mathematics and Teaching of Mathematics and Computer Science in 1998 from Kiev Taras Shevchenko University, and PhD in Computing Science in 2003 from University of Newcastle upon Tyne.

From September 2005 Victor is a Royal Academy of Engineering/EPSRC Post-doctoral Research Fellow, working on the DAVAC project.

Interests: model checking of Petri nets, Petri net unfolding techniques, self-timed (asynchronous) circuits.

Suggested keywords

PETRI NET,
TRANSITION INSERTION,
TRANSFORMATION,
PETRI NET UNFOLDING,
ENCODING CONFLICT

Behaviour-Preserving Transition Insertions in Unfolding Prefixes

Victor Khomenko

School of Computing Science, University of Newcastle
Newcastle upon Tyne, NE1 7RU, U.K.
E-mail: Victor.Khomenko@ncl.ac.uk

Abstract. Some design methods based on Petri nets modify the original specification by behaviour-preserving insertion of new transitions. If the unfolding prefix is used to analyse the net, it has to be re-unfolded after each modification, which is detrimental for the overall performance.
The approach presented in this paper applies the transformations directly to the unfolding prefix, thus avoiding re-unfolding. This also helps in visualisation, since the application of the transformation directly to the prefix changes it in a way that was 'intuitively expected' by the user, while re-unfolding can dramatically change the shape of the prefix. Moreover, rigourous validity checks for several kinds of transition insertions are developed. These checks are performed *on the original unfolding prefix,* so one never has to backtrack due to the choice of a transformation which does not preserve the behaviour.
Keywords: Petri net, transition insertion, transformation, Petri net unfolding, encoding conflict.

1 Introduction

Some design methods based on Petri nets modify the original specification by behaviour-preserving insertion of new transitions. For example, Signal Transition Graphs (STGs) are a formalism widely used for describing the behaviour of asynchronous control circuits. Typically, they are used as a specification language for the synthesis of such circuits [2, 6, 18]. STGs are a class of interpreted Petri nets, in which transitions are labelled with the names of rising and falling edges of circuit signals. In the discussion below, though we have in mind a particular application, viz. synthesis of asynchronous circuits from STG specification, almost all the developed techniques and algorithms are not specific to this application domain and suitable for general Petri nets.

Circuit synthesis based on STGs involves: (i) checking the necessary and sufficient conditions for the STG's implementability as a logic circuit; (ii) modifying, if necessary, the initial STG to make it implementable; and (iii) finding an appropriate Boolean cover for the next-state function of each output and internal signals, and obtaining them in the form of Boolean equations for the logic gates of the circuit.

Step (i) of this process may detect *state encoding conflicts*, which occur when semantically different (i.e., enabling different sets of output signals) reachable markings of the STG have the same binary *encoding*, i.e., the binary vector containing the value of each signal in the circuit, as illustrated in Fig. 1(a,b). A specification containing such encoding conflicts is not directly implementable: intuitively, at the implementation level the only information available to the circuit is the encoding, and so it is unable distinguish between the conflicting states.

To proceed with the synthesis, one first has to *resolve* the encoding conflicts (step (ii) of the process), which is usually done by adding one or more new *internal* signals helping to distinguish between conflicting states, as illustrated in Fig. 1(c,d). Hence, the original STG has to be modified by insertion of new transitions, in such a way that its 'external' behaviour does not change. Intuitively, insertion of new signals extends the encoding vector, introducing thus additional 'memory' helping the circuit to trace the current state.

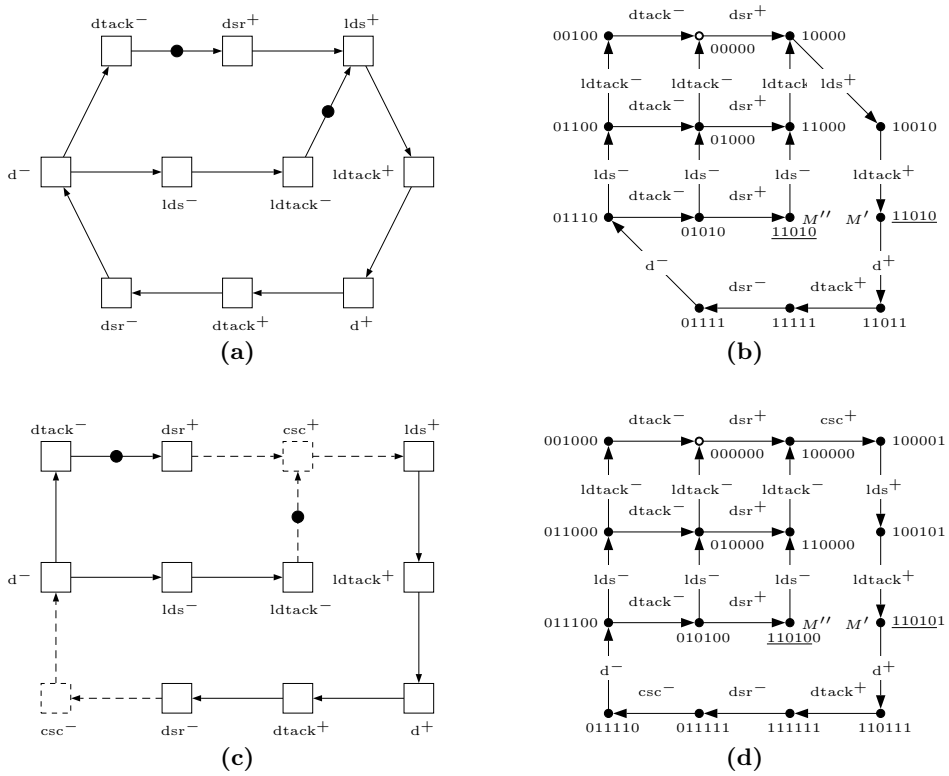


Fig. 1. An STG modelling a simplified VME bus controller **(a)**, its state graph with a CSC conflict between two states **(b)**, a modified STG where the CSC conflict has been resolved by adding a new signal *csc* **(c)**, and its state graph **(d)**. The order of signals in the binary encodings is: *dsr*, *ldtack*, *dtack*, *lds*, *d*[, *csc*].

One of the commonly used STG-based synthesis tools, PETRIFY [4, 6], performs all of these steps automatically, after first constructing the state graph (in the form of a BDD [1, 13]) of the initial STG specification. While the state-based approach is relatively well-studied, the issue of computational complexity for highly concurrent STGs is quite serious due to the state space explosion problem. This puts practical bounds on the size of control circuits that can be synthesised using such techniques, which are often restrictive, especially if the STG models are not constructed manually by a designer but rather generated automatically from high-level hardware descriptions.

In order to alleviate this problem, Petri net analysis techniques based on causal partial order semantics, in the form of Petri net unfoldings, were applied to circuit synthesis. Since in practice STGs usually exhibit a lot of concurrency, but have rather few choice points, their complete unfolding prefixes are often exponentially smaller than the corresponding state graphs; in fact, in many of the experiments conducted in [11] they are just slightly bigger then the original STGs themselves. Therefore, unfolding prefixes are well-suited for both visualisation of an STG's behaviour and alleviating the state space explosion problem. The papers [11, 12, 14] present a complete design flow for complex-gate logic synthesis based on Petri net unfoldings, which completely avoids generating the state graph, and hence has significant advantage both in memory consumption and in execution time, without affecting the quality of the solutions. Moreover, unfoldings are much more visual then state graphs (the latter are hard to understand due to their large sizes and the tendency to obscure causal relationships and concurrency between the events), which enhances the interaction with the user.

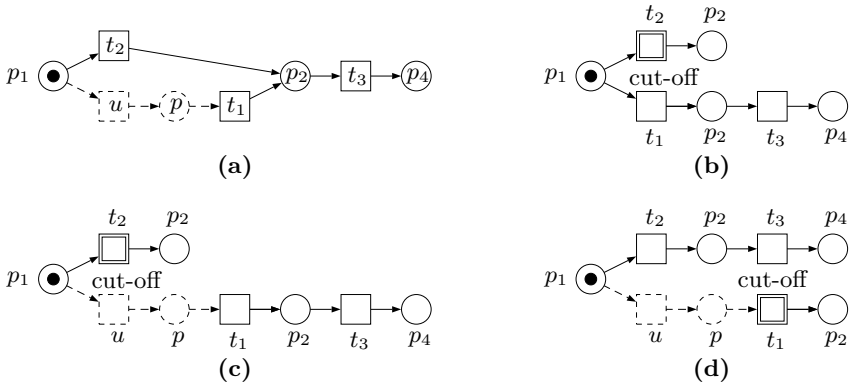


Fig. 2. A Petri net with the transformation shown in dashed lines **(a)**, its unfolding prefix **(b)**, the 'intuitively expected' unfolding prefix of the modified net **(c)**, and the result of re-unfolding **(d)**. The unfolding algorithm proposed in [8] was used for **(b,d)**.

Arguably, the most difficult task in the complex-gate logic synthesis from STGs is resolution of encoding conflicts, which is usually done by signal insertion. This is the only part of the design flow presented in [11, 12, 14] which may require human intervention. In fact, the techniques presented in [14] are targeted at facilitating interaction with the user, by developing a method for visualisation of encoding conflicts.

The tool described in [14] works as follows. First, the STG is unfolded and the encoding conflicts are computed and visualised. Then a set of potentially useful signal insertions is computed and arranged according to a certain cost function. Then the user selects one of these transformations, the STG is modified and the process is repeated until all the encoding conflicts are eliminated. It is currently the responsibility of the user to ensure that the selected transformation is valid — although some validity checks are performed by the tool, it does not guarantee the correctness. Moreover, some of these correctness checks are performed *after* the STG has been modified and re-unfolded, i.e., the tool has to backtrack if the chosen transformation happens to be incorrect (e.g., due to the user's mistake).

The approach presented in this paper improves that in [14] in several ways. First, it applies the transformation not only to the STG, but also directly to the unfolding prefix, thus avoiding re-unfolding on each step of the method. This also helps in visualisation, since the application of the transformation directly to the prefix changes it in a way that was 'intuitively expected' by the designer, while re-unfolding of the modified STG can dramatically change the shape of the prefix (due to different events being declared cut-off, as illustrated in Fig. 2) to which the designer got 'used to'. Moreover, rigourous checks of correctness are developed. These checks are performed *on the original unfolding prefix,* so the algorithm never has to backtrack due to the choice of an incorrect transformation.

Also, there are some problem-specific advantages. In general, not all the encoding conflicts are resolved by a single transformation (hence the need for multiple iterations in the approach in [14]). If the shape of the prefix has changed only slightly and in a predictable way, the unresolved conflicts computed for the original prefix can be transferred to the modified one, which is not generally possible with re-unfolding. This may considerably improve the efficiency of the method.

It should be noted that performing the transformations directly on the prefix is not trivial (in fact, as shown below, naïve algorithms are incorrect), since one has to guarantee the completeness of the resulting prefix. The main difficulty comes from the need to look *beyond cut-off events* of the prefix. Though the idea of transforming the prefix is not new, to our knowledge, this is the first time it is done with a rigourous proof of correctness. In fact, since the transformed prefix can be quite different from that obtained by unfolding the modified

STG, it is complete in a different sense, and to justify the proposed approach we appeal to rather advanced results from the unfolding theory, viz. *canonical prefixes* [10].

2 Basic Notions

In this section, we first present basic definitions concerning Petri nets, and then recall notions related to net unfoldings and canonical prefixes (see also [7–10, 15]).

2.1 Petri nets

A *net* is a triple $N \stackrel{\text{df}}{=} (P, T, F)$ such that P and T are disjoint sets of respectively *places* and *transitions*, and $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*. A *marking* of N is a multiset M of places, i.e., $M : P \to \mathbb{N} = \{0, 1, 2, \ldots\}$. We adopt the standard rules about drawing nets, viz. places are represented as circles, transitions as boxes, the flow relation by arcs, and markings are shown by placing tokens within circles. In addition, the following short-hand notation is used: a transition can be connected directly to another transition if the place 'in the middle of the arc' has exactly one incoming and one outgoing arc (see, e.g., Fig. 1(a,c)). As usual, ${}^\bullet z \stackrel{\text{df}}{=} \{y \mid (y, z) \in F\}$ and $z^\bullet \stackrel{\text{df}}{=} \{y \mid (z, y) \in F\}$ denote the *pre-* and *postset* of $z \in P \cup T$, and we define ${}^\bullet Z \stackrel{\text{df}}{=} \bigcup_{z \in Z} {}^\bullet z$ and $Z^\bullet \stackrel{\text{df}}{=} \bigcup_{z \in Z} z^\bullet$, for all $Z \subseteq P \cup T$. We will assume that ${}^\bullet t \neq \emptyset$, for every $t \in T$. N is *finite* if $P \cup T$ is finite, and *infinite* otherwise.

A *net system* is a tuple $\Sigma \stackrel{\text{df}}{=} (P_\Sigma, T_\Sigma, F_\Sigma, M_\Sigma)$ where $(P_\Sigma, T_\Sigma, F_\Sigma)$ is a finite net and M_Σ is an *initial* marking. A transition $t \in T_\Sigma$ is *enabled* at a marking M, denoted $M[t\rangle$, if, for every $p \in {}^\bullet t$, $M(p) \geq 1$. Such a transition can be *executed* or *fired*, leading to the marking $M' \stackrel{\text{df}}{=} M - {}^\bullet t + t^\bullet$, where '$-$' and '$+$' stand for the multiset difference and sum respectively. We denote this by $M[t\rangle M'$. A finite or infinite sequence $\sigma = t_1 t_2 t_3 \ldots$ of transitions is an *execution* from a marking M, denoted $M[\sigma\rangle$, if $M[t_1\rangle M'$ and $\sigma' = t_2 t_3 \ldots$ is an execution from M'. Moreover, σ is an execution of Σ if $M_\Sigma[\sigma\rangle$. For a transition $t \in T_\Sigma$ and an execution σ we will denote by $\#_t\sigma$ the number of occurrences of t in σ.

The set of *reachable* markings of Σ is the smallest (w.r.t. $\subset$) set containing M_Σ and such that if M is reachable and $M[t\rangle M'$ (for some $t \in T_\Sigma$) then M' is reachable. A transition is *dead* if no reachable marking enables it. A transition is *live* if from any reachable marking M there is an execution containing it. (Note that being live is a stronger property than being non-dead.)

A net system Σ is *k-bounded* if, for every reachable marking M and every place $p \in P_\Sigma$, $M(p) \leq k$, *safe* if it is 1-bounded, and *bounded* if it is k-bounded for some $k \in \mathbb{N}$. The set of reachable markings of Σ is finite iff Σ is bounded.

2.2 Branching processes and canonical prefixes

A *finite and complete unfolding prefix* of a Petri net Σ is a finite acyclic net which implicitly represents all the reachable states of Σ together with transitions enabled at those states. Intuitively, it can be obtained through *unfolding* Σ, by successive firings of transitions, under the following assumptions: (a) for each new firing a fresh transition (called an *event*) is generated; (b) for each newly produced token a fresh place (called a *condition*) is generated. The unfolding is infinite whenever Σ has an infinite run; however, if Σ has finitely many reachable states then the unfolding eventually starts to repeat itself and can be truncated (by identifying a set E_{cut} of *cut-off* events beyond which the prefix is not generated) without loss of information, yielding a finite and complete prefix. Due to its structural properties (such as acyclicity), the reachable markings of Σ can be represented using *configurations* of any of its complete prefixes. Intuitively, a configuration is a partial-order execution, i.e., an execution where the order of firing of some of the events is not important.

Efficient algorithms exist for building finite and complete prefixes [8, 9], which ensure that the number of non-cut-off events in the resulting prefix never exceeds the number of reachable states of Σ. In fact, complete prefixes are often exponentially smaller than the corresponding state graphs, especially for highly concurrent Petri nets, because they represent concurrency directly rather than by multidimensional 'diamonds' as it is done in state graphs. For example, if the original Petri net consists of 100 transitions which can fire once in parallel, the state graph will be a 100-dimensional hypercube with 2^{100} vertices, whereas the complete prefix will coincide with the net itself. The experimental results in [11] demonstrate that high levels of compression can indeed be achieved in practice.

Formally, two nodes (places or transitions), y and y', of a net $N = (P, T, F)$ are *in conflict*, denoted by $y \# y'$, if there are distinct transitions $t, t' \in T$ such that ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$ and (t, y) and (t', y') are in the reflexive transitive closure of the flow relation F, denoted by $\preceq$. A node y is in *self-conflict* if $y \# y$.

An *occurrence net* is a net $ON \stackrel{\text{df}}{=} (B, E, G)$, where B is the set of *conditions* (places) and E is the set of *events* (transitions), satisfying the following: ON is acyclic (i.e., $\preceq$ is a partial order); for every $b \in B$, $|{}^\bullet b| \leq 1$; for every $y \in B \cup E$, $\neg(y \# y)$ and there are finitely many y' such that $y' \prec y$, where $\prec$ denotes the transitive closure of G. $Min(ON)$ will denote the set of minimal (w.r.t. $\prec$) elements of $B \cup E$. The relation $\prec$ is the *causality relation*. Two nodes are *concurrent*, denoted $y \parallel y'$, if neither $y \# y'$ nor $y \preceq y'$ nor $y' \preceq y$.

A *homomorphism* from an occurrence net ON to a net system Σ is a mapping $h : B \cup E \to P_\Sigma \cup T_\Sigma$ such that: $h(B) \subseteq P_\Sigma$ and $h(E) \subseteq T_\Sigma$ (conditions are mapped to places, and events to transitions); for all $e \in E$, the restriction of h to ${}^\bullet e$ is a bijection between ${}^\bullet e$ and ${}^\bullet h(e)$ and the restriction of h to $e^\bullet$ is a bijection between $e^\bullet$ and $h(e)^\bullet$ (transition environments are preserved); the restriction of h to $Min(ON)$ is a bijection between $Min(ON)$ and M_Σ (minimal conditions correspond to the initial marking); and for all $e, f \in E$, if ${}^\bullet e = {}^\bullet f$ and $h(e) = h(f)$ then $e = f$ (there is no redundancy). A *branching process* of Σ is a tuple $\pi_\Sigma \stackrel{\text{df}}{=} (B_{\pi_\Sigma}, E_{\pi_\Sigma}, G_{\pi_\Sigma}, h_{\pi_\Sigma})$ such that $(B_{\pi_\Sigma}, E_{\pi_\Sigma}, G_{\pi_\Sigma})$ is an occurrence net and h_{π_Σ} is a homomorphism from it to Σ. If a node $x \in B_{\pi_\Sigma} \cup E_{\pi_\Sigma}$ is such that $h_{\pi_\Sigma}(x) = y \in P_\Sigma \cup T_\Sigma$, then we will often refer to x as being *y-labelled* or as an *instance of y*.

A branching process $\pi'_\Sigma = (B_{\pi'_\Sigma}, E_{\pi'_\Sigma}, G_{\pi'_\Sigma}, h_{\pi'_\Sigma})$ of Σ is a *prefix* of π_Σ, denoted $\pi'_\Sigma \sqsubseteq \pi_\Sigma$, if $(B_{\pi'_\Sigma}, E_{\pi'_\Sigma}, G_{\pi'_\Sigma})$ is a subnet of $(B_{\pi_\Sigma}, E_{\pi_\Sigma}, G_{\pi_\Sigma})$ containing all minimal elements and such that: if $e \in E_{\pi'_\Sigma}$ and $(b, e) \in G_{\pi_\Sigma}$ or $(e, b) \in G_{\pi_\Sigma}$ then $b \in B_{\pi'_\Sigma}$; if $b \in B_{\pi'_\Sigma}$ and $(e, b) \in G_{\pi_\Sigma}$ then $e \in E_{\pi'_\Sigma}$; and $h_{\pi'_\Sigma}$ is the restriction of h_{π_Σ} to $B_{\pi'_\Sigma} \cup E_{\pi'_\Sigma}$. For each Σ there exists a unique (up to isomorphism) maximal (w.r.t. $\sqsubseteq$) branching process Unf^{max}_Σ, called the *unfolding* of Σ. To simplify the notation, we will write h_Σ instead of h_{π_Σ}; this is justified since $h_{\pi_\Sigma}(x)$ is the same in any branching process of Σ containing x, in particular, one can always refer to Unf^{max}_Σ.

An example of a safe net system and two of its branching prefixes is shown in Fig. 3, where the homomorphism h is indicated by the labels of the nodes. The process in Fig. 3(b) is a prefix of that in Fig. 3(c).

Configurations and cuts A *configuration* of a branching process π_Σ is a finite set of events $C \subseteq E_{\pi_\Sigma}$ such that for all $e, f \in C$, $\neg(e \# f)$ and, for every $e \in C$, $f \prec e$ implies $f \in C$. For every event $e \in E_{\pi_\Sigma}$, the configuration $[e]_\Sigma \stackrel{\text{df}}{=} \{f \mid f \preceq e\}$ is called the *local configuration* of e. Moreover, for a set of events E' we denote by $C \oplus E'$ the fact that $C \cup E'$ is a configuration and $C \cap E' = \emptyset$. Such a set is a *suffix* of C, and the configuration $C \oplus E'$ is an *extension* of C. We will write $C \oplus e$ instead of $C \oplus \{e\}$. For a transition $t \in T_\Sigma$ and a configuration C of π_Σ we will denote by $\#_t C$ the number of t-labelled events in C.

A set of events E' is *downward-closed* if all causal predecessors of the events in E' also belong to E'. Such a set *induces* a unique branching process π_Σ whose events are exactly the events in E', and whose conditions are the conditions incident to the events in E' together with the causally minimal conditions.

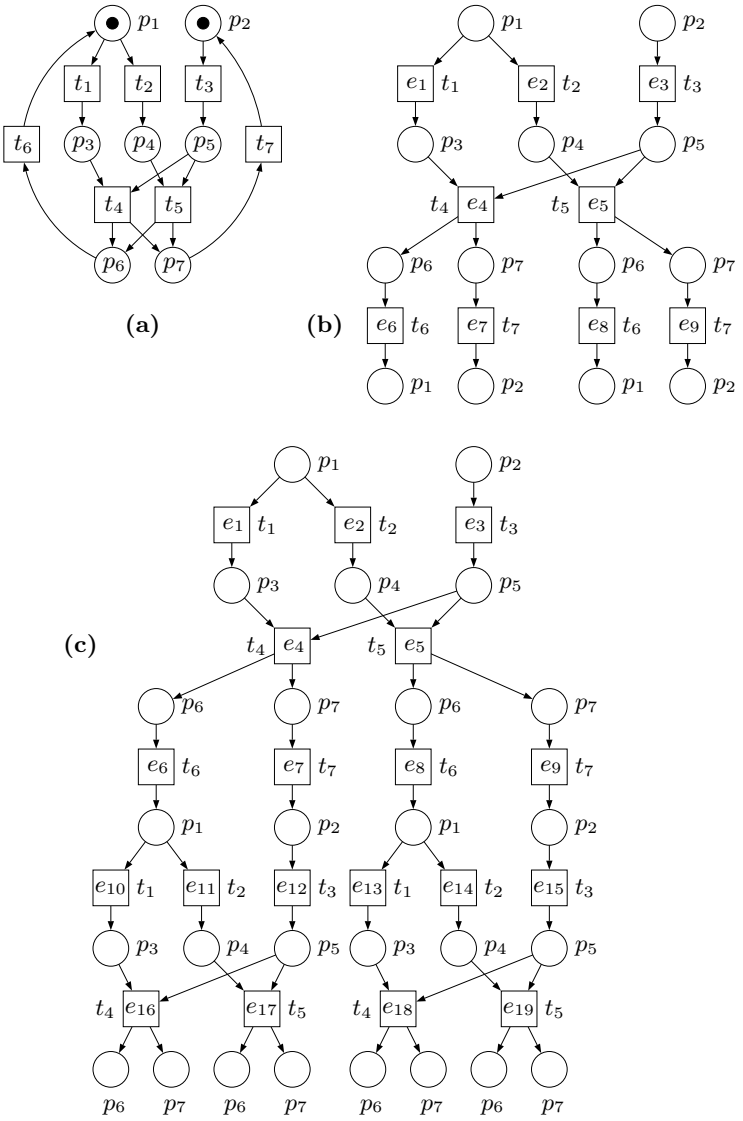


Fig. 3. A net system **(a)** and two of its branching processes **(b,c)**.

A set of conditions B' such that for all distinct $b, b' \in B'$, $b \parallel b'$, is called a *co-set*. A *cut* is a maximal (w.r.t. $\subset$) co-set. Every marking reachable from $Min(ON)$ is a cut. If C is a configuration of π_Σ then the set

$$Cut_\Sigma(C) \stackrel{\text{df}}{=} \Big(Min(ON) \cup C^\bullet\Big) \setminus {}^\bullet C$$

is a cut; moreover, the multiset of places $Mark_\Sigma(C) \stackrel{\text{df}}{=} h_\Sigma(Cut_\Sigma(C))$ is a reachable marking of Σ, called the *final marking* of C. A marking M of Σ is *represented* in π_Σ if there is a configuration C of π_Σ such that $M = Mark_\Sigma(C)$. Every marking represented in π_Σ is reachable in the original net system Σ, and every reachable marking of Σ is represented in Unf^{max}_Σ.

Note that the notations $[\cdot]_\Sigma$, $Cut_\Sigma(\cdot)$ and $Mark_\Sigma(\cdot)$ differ from the conventional ones by the presence of subscripts. This is useful in our settings since we modify unfolding prefixes directly when the original Petri net is modified, i.e., the same event e may belong to branching processes of two different Petri nets, viz. the original and modified ones, and the subscript

is needed to distinguish between them. That we use Σ rather than π_Σ as the subscripts is justified in the view of the fact that the denoted objects are the same in any branching process of Σ containing the necessary events; in particular, one can always refer to Unf_Σ^{max}.

Cutting context There exist different methods of truncating Petri net unfoldings. The differences are related to the kind of information about the original unfolding one wants to preserve in the prefix, as well as to the choice between using either only local configurations (which can improve the running time of an algorithm), or all configurations (which can result in a smaller prefix).

In order to cope with different variants of the technique for truncating unfoldings, we use an abstract parametric model developed in [10]. The first parameter will determine the information we intend to preserve in a complete prefix (in the standard case, this is the set of reachable markings). The main idea behind it is to speak about configurations of Unf_Σ^{max} rather than reachable markings of Σ. Formally, the information to be preserved corresponds to the equivalence classes of some equivalence relation $\approx$ on the configurations of Unf_Σ^{max}. The other parameters are more technical: they specify the circumstances under which an event can be designated as a cut-off event.

Definition 1 (Cutting context). *A triple* $\Theta \stackrel{\text{df}}{=} \left(\approx, \lhd, \{\mathcal{C}_e\}_{e \in E_{Unf_\Sigma^{max}}}\right)$ *is a* cutting context *if:*

1. $\approx$ *is an equivalence relation on the configurations of* Unf_Σ^{max}.
2. $\lhd$*, called an* adequate *order, is a strict well-founded partial order on the configurations of* Unf_Σ^{max} *refining* $\subset$*, i.e.,* $C' \subset C''$ *implies* $C' \lhd C''$.
3. $\approx$ *and* $\lhd$ *are* preserved by finite extensions*, i.e., for every pair of configurations* $C' \approx C''$*, and for every suffix* E' *of* C'*, there exists a finite suffix* E'' *of* C'' *such that*
   (a) $C'' \oplus E'' \approx C' \oplus E'$*, and*
   (b) *if* $C'' \lhd C'$ *then* $C'' \oplus E'' \lhd C' \oplus E'$.
4. $\{\mathcal{C}_e\}_{e \in E_{Unf_\Sigma^{max}}}$ *is a family of sets of configurations of* Unf_Σ^{max}. ◊

The main idea behind the adequate order is to specify which configurations will be preserved in the complete prefix; it turns out that all $\lhd$-minimal configurations in each equivalence class of $\approx$ will be preserved. The last parameter is needed to specify the set of configurations used later to decide whether an event can be designated as a cut-off event. For example, $\mathcal{C}_e$ may contain all configurations of Unf_Σ^{max}, or, as it is usually the case in practice, only the local ones.

We will write $e \lhd f$ instead of $[e]_\Sigma \lhd [f]_\Sigma$, provided that this does not create an ambiguity. Clearly, $\lhd$ is a well-founded partial order on the set of events. Hence, we can use Noetherian induction (see [3]) for definitions and proofs, i.e., it suffices to define or prove something for an event under the assumption that it has already been defined or proven for all its $\lhd$-predecessors.

In this paper we assume that the first component of the cutting context is the equivalence of final markings, defined as $C' \approx_{mar} C''$ iff $Mark_\Sigma(C') = Mark_\Sigma(C'')$. The first time the unfolding prefix is built, some fixed cutting context, e.g., $\Theta_{ERV} \stackrel{\text{df}}{=} \left(\approx_{mar}, \lhd_{erv}, \{\mathcal{C}_e\}_{e \in E_{Unf_\Sigma^{max}}}\right)$ — the cutting context corresponding to the framework used by Esparza, Römer and Vogler in [8], can be used. Here $\lhd_{erv}$ the total adequate order for safe Petri nets proposed in [8] and, for each $e \in E_{Unf_\Sigma^{max}}$, $\mathcal{C}_e$ comprises the local configurations of Unf_Σ^{max}. However, as transformations are performed, *all* components of the cutting context can change (the equivalence relation also changes, albeit for the trivial reason that the modified net has additional places and/or transitions).

Completeness of branching processes We now introduce a notion of completeness for branching processes.

Definition 2 (Completeness). *A branching process π_Σ is* complete w.r.t. a set E_{cut} *of events of Unf^{max}_Σ if the following hold:*

1. *If C is a configuration of Unf^{max}_Σ then there is a configuration C' of π_Σ such that $C' \cap E_{cut} = \emptyset$ and $C \approx C'$.*
2. *If C is a configuration of π_Σ such that $C \cap E_{cut} = \emptyset$, and e is an event such that $C \oplus e$ is a configuration of Unf^{max}_Σ, then $C \oplus e$ is a configuration of π_Σ.*

A branching process π_Σ is complete *if it is complete w.r.t. some set E_{cut}.* $\Diamond$

Note that, π_Σ remains complete after removing all events e for which $([e]_\Sigma \setminus \{e\}) \cap E_{cut} \neq \emptyset$, and so, without loss of generality, one can assume that E_{cut} contains only causally maximal events of π_Σ. Note also that the last definition depends only on the equivalence $\approx$, and not on the other components of the cutting context.

Canonical prefix Now we define *static* cut-off events, without reference to any unfolding algorithm (hence the term 'static'), together with *feasible* events, which are precisely those events whose causal predecessors are not cut-off events, and as such must be included in the prefix determined by the static cut-off events.

Definition 3 (Feasible and cut-off events). *The set of* feasible *events, denoted by $fsble_\Theta$, and the set of* static cut-off *events, denoted by cut_Θ, are two sets of events of Unf^{max}_Σ defined inductively, in the following way:*

1. *An event e is a feasible event if $([e]_\Sigma \setminus \{e\}) \cap cut_\Theta = \emptyset$.*
2. *An event e is a static cut-off event if it is feasible, and there is a configuration $C \in \mathcal{C}_e$ such that $C \subseteq fsble_\Theta \setminus cut_\Theta$, $C \approx [e]_\Sigma$, and $C \lhd [e]_\Sigma$. In what follows, any C satisfying these conditions will be called a* corresponding *configuration of e.* $\Diamond$

The sets $fsble_\Theta$ and cut_Θ are well-defined sets due to Noetherian induction [10].

Once we have defined the feasible events, the notion of the canonical prefix arises quite naturally, after observing that $fsble_\Theta$ is a downward-closed set of events.

Definition 4 (Canonical prefix). *The branching process $Pref^\Theta_\Sigma$ induced by the set of events $fsble_\Theta$ is called the* canonical prefix *of Unf^{max}_Σ.* $\Diamond$

Note that $Pref^\Theta_\Sigma$ is uniquely determined by the cutting context Θ.

Several fundamental properties of $Pref^\Theta_\Sigma$ are proven in [10]. In particular, $Pref^\Theta_\Sigma$ is always complete w.r.t. $E_{cut} = cut_\Theta$, and it is finite if $\approx$ has finitely many equivalence classes (in the case of $\approx_{mar}$ the equivalence classes correspond to the reachable markings, i.e., this condition is equivalent to the boundedness of the Petri net) and, for each $e \in E_{Unf^{max}_\Sigma}$, $\mathcal{C}_e$ contains all the local configurations of Unf^{max}_Σ.

3 Transformations

In this paper, we are primarily interested in *SB-preserving* transformations, i.e., transformations preserving safeness and behaviour (in the sense that the original and the transformed Petri nets are bisimular, provided that the newly inserted transitions are considered silent) of the Petri net. This section describes several kinds of transition insertions.

3.1 Sequential pre-insertion

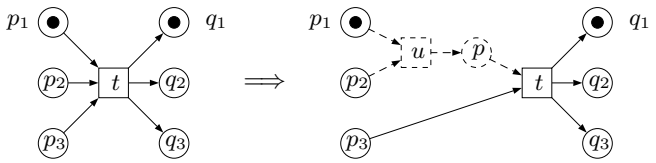
A sequential pre-insertion is essentially a generalised transition splitting, and is formally defined as follows.

Definition 5 (Sequential pre-insertion). *Given a Petri net $\Sigma = (P_\Sigma, T_\Sigma, F_\Sigma, M_\Sigma)$, a transition $t \in T_\Sigma$ and a non-empty set of places $S \subseteq {}^\bullet t$, the* sequential pre-insertion $S \wr t$ *is the transformation yielding the Petri net $\Sigma^u = (P_{\Sigma^u}, T_{\Sigma^u}, F_{\Sigma^u}, M_{\Sigma^u})$, where*

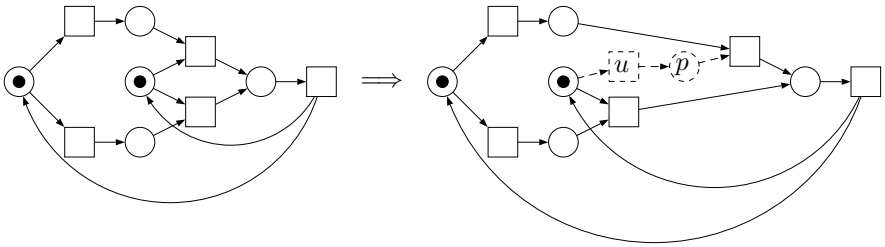
- $P_{\Sigma^u} \stackrel{\text{df}}{=} P_\Sigma \cup \{p\}$, *where $p \notin P_\Sigma \cup T_\Sigma$ is a new place;*
- $T_{\Sigma^u} \stackrel{\text{df}}{=} T_\Sigma \cup \{u\}$, *where $u \notin P_\Sigma \cup T_\Sigma \cup \{p\}$ is a new transition;*
- $F_{\Sigma^u} \stackrel{\text{df}}{=} (F_\Sigma \setminus \{(s,t) | s \in S\}) \cup \{(s,u) | s \in S\} \cup \{(u,p),(p,t)\}$.
- $M_{\Sigma^u} = M_\Sigma$.

We will write $\wr t$ instead of $S \wr t$ if $S = {}^\bullet t$, and $s \wr t$ instead of $\{s\} \wr t$. ◊

The picture below illustrates the sequential pre-insertion $\{p_1, p_2\} \wr t$.



Sequential pre-insertion always preserves safeness and trace-equivalence (the latter is due to the fact that the contraction of the inserted transition is a type-II secure contraction in the sense of [17]). However, in general, the behaviour is not preserved, and so a sequential pre-insertion is not guaranteed to be SB-preserving. In fact, it can introduce deadlocks, as illustrated in the picture below.



Hence, one has to impose additional conditions on the transformation to guarantee that it is SB-preserving. The intuition behind the proposition below is that it is enough to show that the newly inserted transition never 'steals' tokens from the preset of any enabled transition, i.e., its firing cannot disable any other transition.

Proposition 1 (SB-preserving pre-insertions). *Given a safe Petri net Σ and a sequential pre-insertion $S \wr t$ in it such that Σ has no reachable marking $M \supseteq S$ enabling some transition $t' \neq t$ such that $S \cap {}^\bullet t' \neq \emptyset$. Then $S \wr t$ is SB-preserving.*

The condition formulated in this proposition is a simple reachability property which can be efficiently tested on the original unfolding prefix *before the transformation*: one has to find for each transition $t' \in S^\bullet \setminus \{t\}$ a reachable marking M covering $S \cup {}^\bullet t'$, which can be done efficiently using, e.g., the techniques described in [9]. Moreover, in certain special cases, e.g., if $S^\bullet = \{t\}$, this condition is always satisfied and hence the reachability analysis can be skipped.

It should be noted that the above proposition is a sufficient but not a necessary condition: for example, a sequential pre-insertion of the form $\wr t$ is always SB-preserving, even if the inserted transition can 'steal' tokens from other enabled transitions.

3.2 Sequential post-insertion

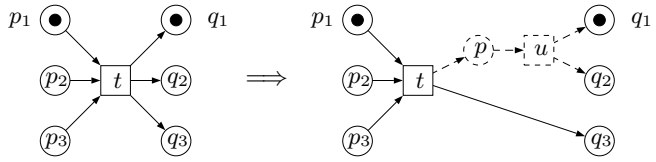
Similarly to sequential pre-insertion, sequential post-insertion is also a generalisation of transition splitting, and it is formally defined as follows.

Definition 6 (Sequential post-insertion). *Given a Petri net $\Sigma = (P_\Sigma, T_\Sigma, F_\Sigma, M_\Sigma)$, a transition $t \in T_\Sigma$ and a non-empty set of places $S \subseteq t^\bullet$, the* sequential pre-insertion $t \wr S$ *is the transformation yielding the Petri net $\Sigma^u = (P_{\Sigma^u}, T_{\Sigma^u}, F_{\Sigma^u}, M_{\Sigma^u})$, where*

- $P_{\Sigma^u} \stackrel{\text{df}}{=} P_\Sigma \cup \{p\}$, *where $p \notin P_\Sigma \cup T_\Sigma$ is a new place;*
- $T_{\Sigma^u} \stackrel{\text{df}}{=} T_\Sigma \cup \{u\}$, *where $u \notin P_\Sigma \cup T_\Sigma \cup \{p\}$ is a new transition;*
- $F_{\Sigma^u} \stackrel{\text{df}}{=} (F_\Sigma \setminus \{(t,s) | s \in S\}) \cup \{(t,p),(p,u)\} \cup \{(u,s) | s \in S\}$.
- $M_{\Sigma^u} = M_\Sigma$.

We will write $t\wr$ instead of $t \wr S$ if $S = t^\bullet$, and $t \wr s$ instead of $t \wr \{s\}$. ◊

The picture below illustrates the sequential post-insertion $t \wr \{q_1, q_2\}$.



Sequential post-insertions always preserve safeness and behaviour (the latter is due to the fact that the contraction of the inserted transition is a type-I secure contraction in the sense of [17]), and hence are always SB-preserving.

3.3 Concurrent insertion

Concurrent transition insertions can be advantageous when the latency of the Petri net should be minimised. It is defined as follows.

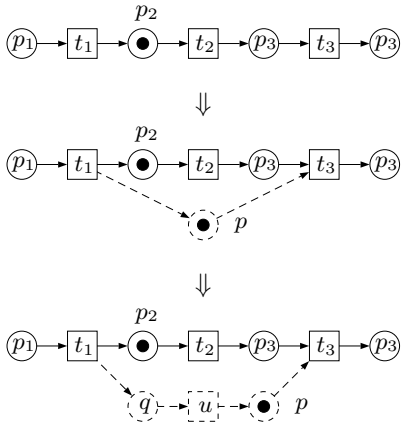
Definition 7 (Concurrent insertion). *Given a Petri net $\Sigma = (P_\Sigma, T_\Sigma, F_\Sigma, M_\Sigma)$, two of its transitions $t, t' \in T_\Sigma$ and an $n \in \mathbb{N}$, the* concurrent insertion $t' \stackrel{\|n}{\longrightarrow} t''$ *is the transformation yielding the Petri net $\Sigma^u = (P_{\Sigma^u}, T_{\Sigma^u}, F_{\Sigma^u}, M^u_\Sigma)$, where*

- $P_{\Sigma^u} \stackrel{\text{df}}{=} P_\Sigma \cup \{p, q\}$, *where $p, q \notin P_\Sigma \cup T_\Sigma$ are two new places;*
- $T_{\Sigma^u} \stackrel{\text{df}}{=} T_\Sigma \cup \{u\}$, *where $u \notin P_\Sigma \cup T_\Sigma \cup \{p, q\}$ is a new transition;*
- $F_{\Sigma^u} \stackrel{\text{df}}{=} F_\Sigma \cup \{(t', q), (q, u), (u, p), (p, t'')\}$;
- M_{Σ^u} *is M_Σ with n tokens on p.*

We will write $t' \stackrel{\|}{\longrightarrow} t''$ instead of $t' \stackrel{\|0}{\longrightarrow} t''$ and $t' \stackrel{\|\bullet}{\longrightarrow} t''$ instead of $t' \stackrel{\|1}{\longrightarrow} t''$. ◊

One can additionally require that t' cannot 'trigger' t'' (otherwise the transformation is essentially a sequential insertion). However, this is not important from the correctness point of view.

A concurrent insertion can be viewed as a two-stage transformation. In the first stage, a new place p with n tokens on it is inserted; this transformation will be denoted $t' \stackrel{\textcircled{n}}{\longrightarrow} t''$ (or $t' \stackrel{\bigcirc}{\longrightarrow} t''$ or $t' \stackrel{\textcircled{\bullet}}{\longrightarrow} t''$ if n is 0 or 1, respectively), and the resulting Petri net will be denoted Σ^p. Then, the sequential post-insertion $t' \wr p$ is applied. The picture below illustrates the concurrent insertion $t_1 \stackrel{\|\bullet}{\longrightarrow} t_3$.



In general, concurrent insertions preserve neither safeness nor behaviour. In fact, safeness is not preserved even if $n = 0$ (e.g., when in the original net t' should fire twice before t'' can become enabled), and deadlocks can be introduced even if $n = 1$ (e.g., when in the original net t'' can fire twice without t' firing). Hence, one has to impose additional conditions on the transformation to guarantee that it is SB-preserving.

Since sequential post-insertions are always SB-preserving, instead of investigating the validity of a concurrent insertion $t' \xrightarrow{\|n} t''$, it is enough to investigate the validity of the corresponding place insertion $t' \xrightarrow{\textcircled{n}} t''$. One can observe that if the inserted by the transformation $t' \xrightarrow{\textcircled{n}} t''$ place is an *implicit* place [16] (i.e., the absence of tokens in it can never be the sole reason of t'' being disabled) then the place insertion preserves the behaviour. Hence, checking that a place insertion $t' \xrightarrow{\textcircled{n}} t''$ is SB-preserving amounts to checking that the newly inserted place is safe and implicit in the resulting Petri net; however, these conditions should be checked *on the original unfolding prefix*.

Given a place insertion $t' \xrightarrow{\textcircled{n}} t''$ and an execution σ of Σ (respectively, a configuration C of Unf_{Σ}^{max}), we define $Tokens(\sigma) \stackrel{\text{df}}{=} n + \#_{t'}\sigma - \#_{t''}\sigma$ (respectively, $Tokens(C) \stackrel{\text{df}}{=} n + \#_{t'}C - \#_{t''}C$). Intuitively, $Tokens(\sigma)$ is the final number of tokens in the newly inserted place (provided that σ is an execution of the modified Petri net as well), i.e., this is the marking equation (see [16]) for this place. Note that $Tokens(\sigma)$ can be negative.

Proposition 2 (SB-preserving place insertions). *Let Σ be a safe Petri nets and $t' \xrightarrow{\textcircled{n}} t''$ be a place insertion in it. Then $t' \xrightarrow{\textcircled{n}} t''$ is SB-preserving iff for any execution σ of Σ, $Tokens(\sigma) \in \{0, 1\}$, or, equivalently, for any configuration C of Unf_{Σ}^{max}, $Tokens(C) \in \{0, 1\}$.*

Proof. ($\Longrightarrow$) If $t' \xrightarrow{\textcircled{n}} t''$ is SB-preserving and the condition is violated, it's easy to build a contradiction.

($\Longleftarrow$) the newly inserted place is easily seen to be safe and implicit in this case, and so $t' \xrightarrow{\textcircled{n}} t''$ is SB-preserving. □

Intuitively, the above proposition requires t' and t'' alternate in any execution σ of Σ, and, if $n = 0$ then t' should precede t'' in σ and if $n = 1$ then t'' should precede t' in σ. Any execution of Σ or configuration of Unf_{Σ}^{max} violating the condition in Proposition 2 will be called *bad*.

Corollary 1 (Number of tokens). *Let Σ be a safe Petri net, $Pref_{\Sigma}^{\Theta}$ be its canonical w.r.t. the cutting context $\Theta = \left(\approx_{mar}, \lhd, \{\mathcal{C}_e\}_{e \in E_{Unf_{\Sigma}^{max}}}\right)$ prefix and $t' \xrightarrow{\textcircled{n}} t''$ be an SB-preserving*

place insertion in Σ. Then either t' and t'' are dead or

$$n = \begin{cases} 1 & \text{if } \#_{t'}[e]_\Sigma = 0 \text{ for some } t''\text{-labelled event } e \text{ in } Pref^\Theta_\Sigma \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

In effect, Corollary 1 states that in an SB-preserving place insertion $t' \xrightarrow{\textcircled{n}} t''$, only t' and t'' need to be specified, and n can be calculated using (1). Note that even if t' and t'' are dead, (1) still can be used to calculate n, since the choice of n does not matter in such a case.

Now we show how the conditions formulated above can be checked using $Pref^\Theta_\Sigma$. The main difficulty is that a bad configuration of Unf^{max}_Σ can contain cut-off events, and so a part of it can be not in $Pref^\Theta_\Sigma$, i.e., one has to look *beyond cut-off events of the prefix*.

The key idea of the algorithm below is to check for each cut-off event e with a corresponding configuration C (note that $[e]_\Sigma \approx_{mar} C$) that after insertion of p the final markings of $[e]_\Sigma$ and C will still be equal, i.e., C will still be a corresponding configuration of e. This amounts to checking that $Tokens([e]_\Sigma) = Tokens(C)$. It turns out that if this condition holds and there is a bad configuration of Unf^{max}_Σ then one can find a bad configuration already in $Pref^\Theta_\Sigma$.

The following algorithm, given t' and t'', checks whether the transformation $t' \xrightarrow{\textcircled{n}} t''$, for some $n \in \{0,1\}$, is SB-preserving (n is also computed if this is the case). The computation is performed using the original prefix (no need to unfold the modified net).

Algorithm 1 (Checking correctness of a place insertion).

Step 1 *Compute n using (1).*
Step 2 *If $Tokens([e]_\Sigma) \notin \{0,1\}$ for some instance e of t' or t'' then reject the transformation and terminate.*
Step 3 *If $Tokens([e]_\Sigma) \neq Tokens(C)$ for some cut-off event e with a corresponding configuration C then reject the transformation and terminate.*
Step 4 *Accept the transformation.*

It should be noted that in general a cut-off event e can have multiple corresponding configurations. However, in practice only one of them is stored with the cut-off event. Hence one can imagine a situation when a cut-off event has several corresponding configurations and the property $Tokens([e]_\Sigma) = Tokens(C)$ holds for some of them but not the others. It turns out that the algorithm rejects a non-SB-preserving transformation no matter which of this configurations was stored with e.

Proposition 3 (Correctness of Algorithm 1). *Let Σ be a safe Petri net, $t', t'' \in T_\Sigma$ and $Pref^\Theta_\Sigma$ be its canonical w.r.t. the cutting context $\Theta = \left(\approx_{mar}, \lhd, \{\mathcal{C}_e\}_{e \in E_{Unf^{max}_\Sigma}}\right)$ prefix. If Algorithm 1 accepts the transformation then the place insertion $t' \xrightarrow{\textcircled{n}} t''$, where $n \in \{0,1\}$ is computed by the algorithm, is SB-preserving.*

Proof.

Claim 1 There are no two concurrent events e', e'' in $Pref^\Theta_\Sigma$ such that $h_\Sigma(e') = h_\Sigma(e'')$.
Follows from the safeness of Σ.

Claim 2 If the algorithm accepts the transformation then there are no two concurrent events e', e'' in $Pref^\Theta_\Sigma$ such that $h_\Sigma(e') = t'$ and $h_\Sigma(e'') = t''$.
Suppose the opposite, i.e., that there are two concurrent events e', e'' in $Pref^\Theta_\Sigma$ such that $h_\Sigma(e') = t'$ and $h_\Sigma(e'') = t''$, and let $C \stackrel{\text{df}}{=} [e']_\Sigma \cap [e'']_\Sigma$. Since $e' \parallel e''$, any event in $[e']_\Sigma \setminus C$ is concurrent to any event in $[e'']_\Sigma \setminus C$. Hence, due to Claim 1, there are no instances of t'' in $[e']_\Sigma \setminus C$ and there are no instances of t' in $[e'']_\Sigma \setminus C$. Therefore, $Tokens([e']_\Sigma) = Tokens(C) + \#_{t'}[e']_\Sigma > Tokens(C)$ and $Tokens([e'']_\Sigma) = Tokens(C) - \#_{t''}[e'']_\Sigma < Tokens(C)$, and so $Tokens([e']_\Sigma) \notin \{0,1\}$ or $Tokens([e'']_\Sigma) \notin \{0,1\}$. Thus the algorithm should have rejected the transformation when processing e' or e'', a contradiction.

To the contrary, suppose that the algorithm has accepted a non-SB-preserving transformation. Then there is a bad configuration $\widehat{C} \oplus \widehat{e}$ of Unf_{Σ}^{max} (perhaps, containing cut-off and post-cut-off events) such that $h_{\Sigma}(e) \in \{t', t''\}$. Since $Pref_{\Sigma}^{\Theta}$ satisfies the condition in Step 3 of the algorithm, it remains canonical w.r.t. the cutting context where $\approx_{mar}$ is replaced by the following equivalence relation $\approx$: $C_1 \approx C_2 \iff (C_1 \approx_{mar} C_2) \wedge Tokens(C_1) = Tokens(C_2)$. Thus, due to the completeness of $Pref_{\Sigma}^{\Theta}$, any minimal w.r.t. $\lhd$ configuration C such that $C \approx \widehat{C}$ is in $Pref_{\Sigma}^{\Theta}$ and contains no cut-off events (such a minimal configuration exists due to well-foundedness of $\lhd$), and there is an event e in $Pref_{\Sigma}^{\Theta}$ such that $h_{\Sigma}(e) = h_{\Sigma}(\widehat{e}) \in \{t', t''\}$ and $C \oplus e$ is a bad configuration (e may be a cut-off event of $Pref_{\Sigma}^{\Theta}$).

Since all the events in $C \setminus [e]_{\Sigma}$ are concurrent to e, by Claims 1 and 2 there are no instances of t' and t'' in $C \setminus [e]_{\Sigma}$, i.e., $Tokens([e]_{\Sigma}) = Tokens(C \oplus e)$. Thus $[e]_{\Sigma}$ is a bad configuration and the algorithm should have rejected the transformation, a contradiction. □

Note that the inverse of this Proposition is, in general, not true, i.e., an SB-preserving transformation may be rejected by the algorithm. However, this is conservative and Proposition 5 below shows that in practically important cases Algorithm 1 is exact.

Proposition 4 (The live case). *If t' or t'' is live and C' and C'' are configurations of Unf_{Σ}^{max} such that $C' \approx_{mar} C''$ and $Tokens(C') \neq Tokens(C'')$ then the place insertion $t' \xrightarrow{\textcircled{n}} t''$ is not SB-preserving.*

Proof. Since $C' \approx_{mar} C''$ and due to the liveness of t' or t'', there are suffixes E' of C' and E'' of C'' such that $h_{\Sigma}(E') = h_{\Sigma}(E'')$ and $\#_{t'}E' + \#_{t''}E' = \#_{t'}E'' + \#_{t''}E'' = 1$. Hence, due to $Tokens(C') \neq Tokens(C'')$, either $C' \oplus E'$ or $C'' \oplus E''$ is bad and so the transformation is not SB-preserving. □

Proposition 5 (Strengthening of Proposition 3 in the live case). *Let Σ be a safe Petri net, t' and t'' be two of its transitions and $Pref_{\Sigma}^{\Theta}$ be its canonical w.r.t. the cutting context $\Theta = \left(\approx_{mar}, \lhd, \{\mathcal{C}_e\}_{e \in E_{Unf_{\Sigma}^{max}}}\right)$ prefix. If t' or t'' is live then Algorithm 1 accepts the place insertion $t' \xrightarrow{\textcircled{n}} t''$, where $n \in \{0, 1\}$ is computed by the algorithm, iff it is SB-preserving.*

Proof. In the view of Proposition 3, it remains to show that if t' or t'' is live then the place insertion rejected by Algorithm 1 is not SB-preserving.

A transformation rejected in Step 2 of the algorithm is trivially not SB-preserving. Suppose the transformation is rejected in Step 3 of the algorithm. Then in $Pref_{\Sigma}^{\Theta}$ there is a cut-off event e with a corresponding configuration C such that $[e]_{\Sigma} \approx_{mar} C$ and $Tokens([e]_{\Sigma}) \neq Tokens(C)$, i.e., $[e]_{\Sigma}$ and C satisfy the conditions of Proposition 4, and so the transformation is not SB-preserving. □

4 Insertions in the prefix

This section explains how to perform transition insertions directly in the prefix, avoiding thus re-unfolding. The obtained prefixes, though complete, are not canonical w.r.t. the cutting context $\Theta = \left(\approx_{mar}, \lhd, \{\mathcal{C}_e\}_{e \in E_{Unf_{\Sigma}^{max}}}\right)$ used to obtain the original prefix. Hence below we define a different cutting context $\Theta^u = \left(\approx_{mar}^{u}, \lhd^u, \{\mathcal{C}_e^u\}_{e \in E_{Unf_{\Sigma^u}^{max}}}\right)$, w.r.t. which the obtained prefixes are canonical. Here $\approx_{mar}^{u}$ the equivalence of final markings of configurations of $Unf_{\Sigma^u}^{max}$; note that it is, technically speaking, different from $\approx_{mar}$ because its domain is different and Σ^u has one more place than Σ.

The proposition below explains the correspondence between the configurations of Unf_{Σ}^{max} and $Unf_{\Sigma^u}^{max}$, assuming that Σ^u is obtained from Σ by a sequential pre- or post-insertion.

Proposition 6 (Correspondence between configurations of Unf^{max}_{Σ} and $Unf^{max}_{\Sigma^u}$). *Let Σ^u is obtained from a safe Petri net Σ by a sequential pre- or post-insertion, C be a configuration of Unf^{max}_{Σ} and C' be a configuration of $Unf^{max}_{\Sigma^u}$.*

1. *The set $\psi(C') \stackrel{\text{df}}{=} \{e \in C' \mid h_{\Sigma^u}(e) \neq u\}$ is a configuration of Unf^{max}_{Σ}.*
2. *There exists a unique configuration $\underline{\varphi}(C)$ of $Unf^{max}_{\Sigma^u}$ such that $\max_{\prec} \underline{\varphi}(C)$ does not contain u-labelled events and $\psi(\underline{\varphi}(C)) = C$. Moreover, there are at most two configurations in $Unf^{max}_{\Sigma^u}$, $\underline{\varphi}(C)$ and $\underline{\varphi}(C) \oplus e$ (where $h_{\Sigma^u}(e) = u$), such that $\psi(\underline{\varphi}(C)) = \psi(\underline{\varphi}(C \oplus e)) = C$. We define by $\overline{\varphi}(C)$ the latter configuration, if it exists, and $\overline{\varphi}(C) \stackrel{\text{df}}{=} \underline{\varphi}(C)$ otherwise.*
3. *either $\underline{\varphi}(\psi(C')) = C'$ or $\underline{\varphi}(\psi(C')) \oplus e = C'$, and either $\overline{\varphi}(\psi(C')) = C'$ or $\overline{\varphi}(\psi(C')) = C' \oplus e$, for some instance e of u.*
4. *If C is local then $\underline{\varphi}(C)$ is local ($\overline{\varphi}(C)$ is not necessarily local, and if C' is local then $\psi(C')$ is not necessarily local).*

Proposition 7 (Cutting context: $\lhd^u$). *Let Σ^u is obtained from a safe Petri net Σ by a sequential pre- or post-insertion and $\lhd$ be an adequate order on the configurations of Unf^{max}_{Σ}. Then the relation $\lhd^u$ between the configurations of $Unf^{max}_{\Sigma^u}$ defined as $C' \lhd^u C''$ iff either $\psi(C') \lhd \psi(C'')$ or $\psi(C') = \psi(C'')$ and $\#_u C' < \#_u C''$ is an adequate order on the configurations of $Unf^{max}_{\Sigma^u}$. Moreover, $\lhd^u$ is total if $\lhd$ is total.*

Proof. It is trivial to show that $\lhd^u$ is a strict well-founded order refining $\subset$. Hence, to prove that $\lhd^u$ is an adequate order it remains to show that $\lhd^u$ is preserved by finite extensions, i.e., if $C' \approx^u_{mar} C''$ and $C' \lhd^u C''$ then for any finite extension E' of C', $C' \oplus E' \lhd^u C'' \oplus E''$ for some finite extension E'' of C''.

It is enough to show this in the case when E' is a singleton $\{e'\}$; the required property follows then by induction. Note that since $C' \approx^u_{mar} C''$, there exists a (unique due to the safeness of Σ^u) possible extension $\{e''\}$ of C'' such that $h_{\Sigma^u}(e') = h_{\Sigma^u}(e'')$. We now show that $C' \oplus e' \lhd^u C'' \oplus e''$.

First, suppose $h_{\Sigma^u}(e') \neq u$. Then $\psi(C' \oplus e') = \psi(C') \oplus e'$ and $\psi(C'' \oplus e'') = \psi(C'') \oplus e''$. Since $C' \lhd^u C''$, either $\psi(C') \lhd \psi(C'')$ and so $\psi(C') \oplus e' \lhd \psi(C'') \oplus e''$ (due to $\lhd$ being an adequate order and hence preserved by finite extensions), or $\psi(C') = \psi(C'')$ and $\#_u C' < \#_u C''$ and so $e' = e''$ (due to the uniqueness of e''), $\psi(C') \oplus e' = \psi(C'') \oplus e''$ and $\#_u(C' \oplus e') = \#_u C' < \#_u C'' = \#_u(C'' \oplus e'')$. In either case $C' \oplus e' \lhd^u C'' \oplus e''$.

Second, suppose $h_{\Sigma^u}(e') = u$. Then $\psi(C' \oplus e') = \psi(C')$ and $\psi(C'' \oplus e'') = \psi(C'')$. Since $C' \lhd^u C''$, either $\psi(C') \lhd \psi(C'')$ and so $\psi(C' \oplus e') \lhd \psi(C'' \oplus e'')$, or $\psi(C') = \psi(C'')$ and $\#_u C' < \#_u C''$ and so $\psi(C' \oplus e') = \psi(C'' \oplus e'')$ and $\#_u(C' \oplus e') = \#_u C' + 1 < \#_u C'' + 1 = \#_u(C'' \oplus e'')$. In either case $C' \oplus e' \lhd^u C'' \oplus e''$.

Hence, $\lhd^u$ is an adequate order. Moreover, if $\lhd$ was total then the totality of $\lhd^u$ easily follows from Proposition 6(1,2). □

4.1 Sequential pre-insertion

Given a sequential pre-insertion $S \wr t$, we now show how to build $Pref^{\Theta^u}_{\Sigma^u}$ from $Pref^{\Theta}_{\Sigma}$. (Note that $S \wr t$ is not assumed to be SB-preserving.) First of all, it should be noted that the naïve algorithm which simply splits each t-labelled event is, in general, incorrect: it can result in an incomplete prefix or even in an object which is not a branching process, as illustrated in Fig. 4 and 5.

Remark 1. The naïve algorithm does work for some important special cases, e.g., for sequential pre-insertions of the form $\wr t$. This follows from the correctness of Algorithm 2 below (Proposition 8) and the observation that due to completeness of the $Pref^{\Theta}_{\Sigma}$ the only appropriate co-sets in Step 1 of this algorithm are the presets of the instances of t.

Below we describe an algorithm based on a different idea.

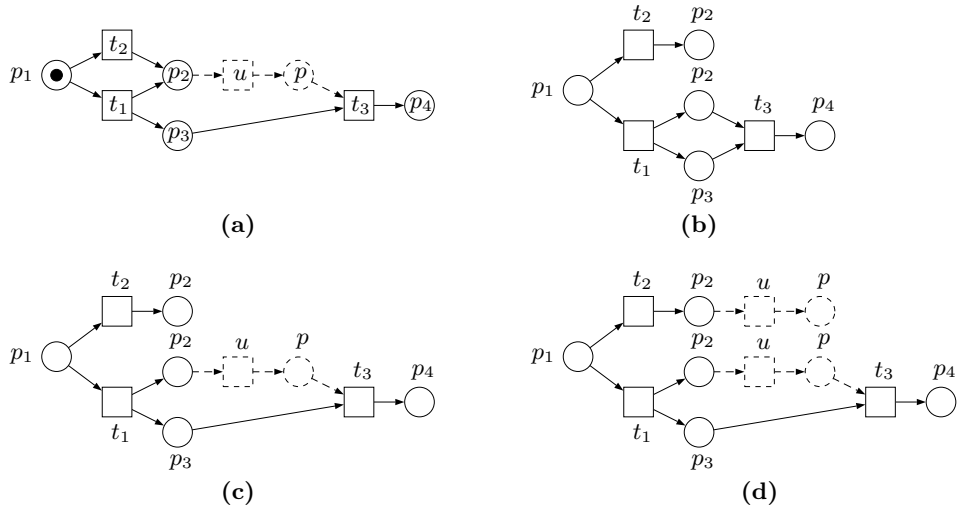


Fig. 4. A Petri net with the transformation shown in dashed lines **(a)**, its unfolding **(b)**, the incomplete branching process obtained by naïve splitting **(c)**, and the complete branching process **(d)**.

Algorithm 2 (Sequential pre-insertion in the prefix).

Step 1 *For each co-set X containing no post-cut-off conditions and such that $h_\Sigma(X) = S$, create an instance of the new transition u, and make X its preset; create also an instance of the new place p, and make it the postset of the inserted transition instance.*

Step 2 *For each t-labelled event e (including cut-off events), let $X \subseteq {}^\bullet e$ be such that $h_\Sigma(X) = S$ (note that X is a co-set); moreover, let f be the unique u-labelled event with the preset X, and c be the p-labelled condition in $f^\bullet$. Remove the conditions in X from the preset of e, and add c there instead.*

Step 3 *For each cut-off event e with a corresponding configuration C, change the corresponding configuration to $\underline{\varphi}(C)$.*

Proposition 8 (Correctness of Algorithm 2). *Let Σ be a safe Petri net and $Pref_\Sigma^\Theta$ be its canonical w.r.t. a cutting context $\Theta = \left(\approx_{mar}, \lhd, \{\mathcal{C}_e\}_{e \in E_{Unf_\Sigma^{max}}}\right)$ prefix. If the Petri net Σ^u is obtained from Σ by the transformation $S \wr t$ then the prefix $Pref_{\Sigma^u}$ computed by Algorithm 2 coincides with the canonical w.r.t. the cutting context $\Theta^u = \left(\approx^u_{mar}, \lhd^u, \{\mathcal{C}^u_e\}_{e \in E_{Unf^{max}_{\Sigma^u}}}\right)$ prefix $Pref_{\Sigma^u}^{\Theta^u}$ of Σ^u, where*

$$\mathcal{C}^u_e \stackrel{\text{df}}{=} \begin{cases} \{\underline{\varphi}(C) \mid C \in \mathcal{C}_e\} & \text{if } h_{\Sigma^u}(e) \neq u \\ \emptyset & \text{if } h_{\Sigma^u}(e) = u. \end{cases}$$

Proof.

Claim 1 The object $Pref_{\Sigma^u}$ produced by Algorithm 2 is a branching process.
One can easily show that $Pref_{\Sigma^u}$ is an occurrence net and its labelling is a homomorphism from it to Σ^u.

Claim 2 If e is a cut-off event of $Pref_{\Sigma^u}$ with a corresponding configuration C then e is causally maximal, $C \approx^u_{mar} [e]_{\Sigma^u}$ and $C \lhd^u [e]_{\Sigma^u}$. Moreover, $Pref_{\Sigma^u}$ cannot be extended without consuming a post-cut-off condition.
The maximality of e is trivial. Since $h_{\Sigma^u}(e) \neq u$, $p \notin Mark_{\Sigma^u}([e]_{\Sigma^u})$. Moreover, since $C = \underline{\varphi}(D)$ for some configuration D of $Pref_\Sigma^\Theta$, $u \notin h_{\Sigma^u}(\max_\prec C)$, and so $p \notin Mark_{\Sigma^u}(C)$. Hence, $C \approx^u_{mar} [e]_{\Sigma^u}$ due to $D \approx_{mar} [e]_\Sigma$, and Proposition 7 implies that $C \lhd^u [e]_{\Sigma^u}$.
Suppose that $Pref_{\Sigma^u}$ can be extended by an event e such that ${}^\bullet e$ contains no post-cut-off conditions. We now consider three cases.

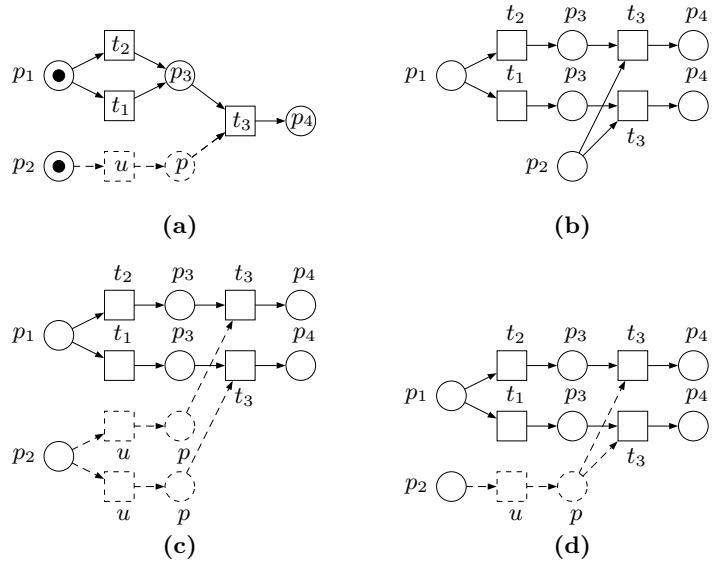


Fig. 5. A Petri net with the transformation shown in dashed lines **(a)**, its unfolding **(b)**, the result of the naïve splitting violating the non-redundancy of nodes condition in the definition of a branching process **(c)**, and the correct unfolding **(d)**.

- If $h_{\Sigma^u}(e) \notin \{u, t\}$ then $Pref^{\Theta}_{\Sigma}$ could be extended by e as well, contradicting the completeness of $Pref^{\Theta}_{\Sigma}$.
- If $h_{\Sigma^u}(e) = u$ then $p \notin h_{\Sigma^u}({}^\bullet e)$ and hence ${}^\bullet e$ is a co-set in $Pref^{\Theta}_{\Sigma}$ containing no cut-off events and such that $h_{\Sigma}({}^\bullet e) = S$. Therefore, the algorithm should have inserted an instance of u with the preset ${}^\bullet e$, a contradiction.
- If $h_{\Sigma^u}(e) = t$ then there is a p-labelled condition $c \in {}^\bullet e$, and hence there is a (unique) u-labelled event $f \in {}^\bullet c$. $X \stackrel{\text{df}}{=} {}^\bullet f \cup ({}^\bullet e \setminus \{c\})$ is a co-set in $Pref^{\Theta}_{\Sigma}$ such that $h_{\Sigma}(X) = {}^\bullet t$. Hence, due to completeness of $Pref^{\Theta}_{\Sigma}$, there is a t-labelled event g in $Pref^{\Theta}_{\Sigma}$ such that ${}^\bullet g = X$. Therefore, g is a t-labelled event in $Pref_{\Sigma^u}$ such that ${}^\bullet g = {}^\bullet e$, a contradiction.

Hence, $Pref_{\Sigma^u}$ cannot be extended without consuming a post-cut-off condition.

By Claim 1, $Pref_{\Sigma^u}$ is a branching process, and we show that $Pref_{\Sigma^u}$ and $Pref^{\Theta^u}_{\Sigma^u}$ co-inside, i.e.,

(i) e is an event of $Pref_{\Sigma^u}$ iff e is an event of $Pref^{\Theta^u}_{\Sigma^u}$; and

(ii) e is cut-off in $Pref_{\Sigma^u}$ iff e is cut-off in $Pref^{\Theta^u}_{\Sigma^u}$.

Due to well-foundedness of $\lhd^u$ (Proposition 7), we can use Noetherian induction on $\lhd^u$. That is, we prove (i)&(ii) assuming that (i)&(ii) holds for every $f \lhd^u e$ (note that Noetherian induction does not require the base case).

Suppose e is in one of $Pref_{\Sigma^u}$, $Pref^{\Theta^u}_{\Sigma^u}$, but not in the other. Then, due to Claim 2 and the completeness of $Pref^{\Theta^u}_{\Sigma^u}$, e is a post-cut-off event in one of them, but not in the other, i.e., there exists an event $f \lhd^u e$ which is cut-off in one of them, but not in the other, which contradicts the induction hypothesis. Hence (i) holds.

Now suppose e is in both branching processes and it is cut-off with a corresponding configuration C in one of them, but not cut-off in the other. Due to the induction hypothesis, all the events in $C \lhd^u [e]_{\Sigma^u}$ are neither cut-off nor post-cut-off in both branching processes. In what follows, we consider two cases.

First, suppose e is cut-off in $Pref_{\Sigma^u}$ but not in $Pref^{\Theta^u}_{\Sigma^u}$. Since the algorithm never declares a u-labelled event cut-off in $Pref_{\Sigma^u}$, $h_{\Sigma^u}(e) \neq u$, and so e was a cut-off event in $Pref^{\Theta}_{\Sigma}$ with a corresponding configuration $D = \psi(C)$. By Claim 2, e satisfies the criteria of a cut-off event in $Pref^{\Theta^u}_{\Sigma^u}$ with a corresponding configuration C, i.e., $C \approx^u_{mar} [e]_{\Sigma^u}$, $C \lhd^u [e]_{\Sigma^u}$ and $C \in \mathcal{C}^u_e$, a contradiction.

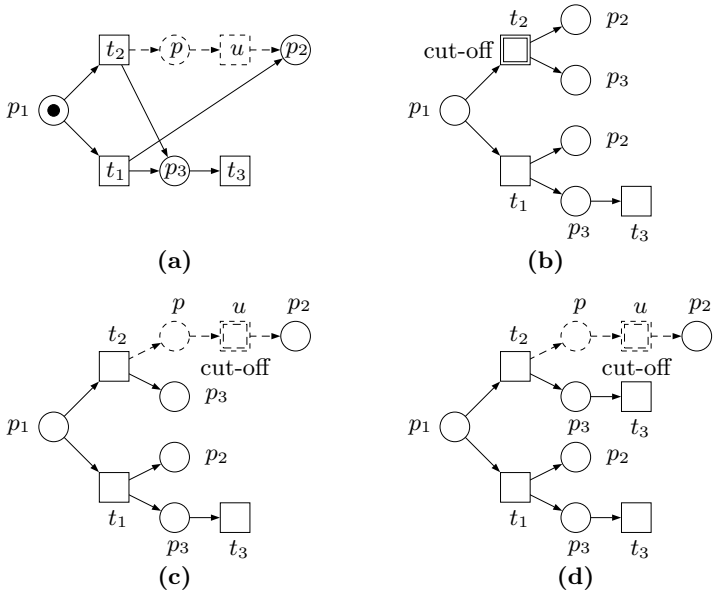


Fig. 6. A Petri net with a sequential post-insertion shown by dashed lines **(a)**, its unfolding **(b)**, the incomplete branching process obtained as the result of a naïve splitting **(c)**, and the complete branching process **(d)**.

Second, suppose e is cut-off in $Pref^{\Theta^u}_{\Sigma^u}$ but not in $Pref_{\Sigma^u}$, i.e., $C \approx^u_{mar} [e]_{\Sigma^u}$, $C \lhd^u [e]_{\Sigma^u}$ and $C \in \mathcal{C}^u_e$. Since $h_{\Sigma^u}(e) \neq u$ (as otherwise $\mathcal{C}^u_e \stackrel{\text{df}}{=} \emptyset$ and so e cannot be cut-off in $Pref^{\Theta^u}_{\Sigma^u}$), one can easily show that e was a cut-off in $Pref^{\Theta}_{\Sigma}$ with a corresponding configuration $D \stackrel{\text{df}}{=} \psi(C)$, i.e., $D \approx_{mar} [e]_{\Sigma}$, $D \lhd^u [e]_{\Sigma}$ and $D \in \mathcal{C}_e$. Hence, the algorithm would have declared e a cut-off in $Pref_{\Sigma^u}$, a contradiction.

Hence (ii) also holds, and we are done. □

4.2 Sequential post-insertion

Given a sequential post-insertion $t \wr S$, we now show how to build $Pref^{\Theta^u}_{\Sigma^u}$ from $Pref^{\Theta}_{\Sigma}$. (Note that sequential post-insertions are always SB-preserving, and so there is no need to check the validity of $t \wr S$.) The algorithm presented below is based on splitting u-labelled events, but special care should be taken when handling cut-off events: a naïve approach may result in an incomplete prefix as illustrated in Fig. 6. If a corresponding configuration C of a cut-off event e has an instance e' of t as a maximal event then e is not split (just its postset is amended), and the corresponding configuration becomes $\underline{\varphi}(C)$ (i.e., the instance of u after e' is not included into it).

In general, it is difficult to guarantee completeness without re-unfolding parts of the prefix, and the algorithm below can sometimes terminate unsuccessfully. In such a case, one either can re-unfold the Petri net or simply not use the transformation (the latter makes sense when there are many alternative transformations to choose from).

Below, a configuration C of $Pref^{\Theta}_{\Sigma}$ is called *u-extendible* if there is a t-labelled event $g \in C$ such that $h_{\Sigma}(g^{\bullet} \cap Cut_{\Sigma}(C)) \subseteq S$. (Intuitively, if C is u-extendible then the configuration $\underline{\varphi}(C)$ of $Pref_{\Sigma^u}$ can be extended by an instance of u).

Algorithm 3 (Sequential post-insertion in the prefix).

Step 1 *If there is a cut-off event e with a corresponding configuration C such that $[e]_{\Sigma}$ is u-extendible and C is not u-extendible then terminate unsuccessfully.*

Step 2 *For each t-labelled event e (including cut-off events): let $X \subseteq e^\bullet$ be the (unique) co-set satisfying $h_\Sigma(X) = S$. In the postset of e replace the conditions in X by a new instance c of p. If e is not a cut-off event then create a new instance of u with the preset $\{c\}$ and the postset X.*

Step 3 *For each cut-off event e of $Pref_\Sigma^\Theta$ with a corresponding configuration C: change the corresponding configuration of e in to $\underline{\varphi}(C)$ if $[e]_\Sigma$ is u-extendible and to $\overline{\varphi}(C)$ otherwise. (In the latter case the corresponding configuration may become non-local, even if C was local.)*

Proposition 9 (Correctness of Algorithm 3). *Let Σ be a safe Petri net and $Pref_\Sigma^\Theta$ be its canonical w.r.t. a cutting context $\Theta = \left(\approx_{mar}, \lhd, \left\{\mathcal{C}_e\right\}_{e \in E_{Unf_\Sigma^{max}}}\right)$ prefix. If the Petri net Σ^u is obtained from Σ by the transformation $t \wr S$ and Algorithm 3 successfully terminates then the computed prefix $Pref_{\Sigma^u}$ coincides with the canonical w.r.t. the cutting context $\Theta^u = \left(\approx^u_{mar}, \lhd^u, \left\{\mathcal{C}^u_e\right\}_{e \in E_{Unf^{max}_{\Sigma^u}}}\right)$ prefix $Pref^{\Theta^u}_{\Sigma^u}$ of Σ^u, where*

$$\mathcal{C}^u_e \stackrel{\text{df}}{=} \begin{cases} \emptyset & \text{if } h_{\Sigma^u}(e) = u \\ \{\underline{\varphi}(C) \mid C \in \mathcal{C}_e\} & \text{if } h_{\Sigma^u}(e) \neq u \text{ and } \psi([e]_{\Sigma^u}) \text{ is } u\text{-extendible} \\ \{\overline{\varphi}(C) \mid C \in \mathcal{C}_e\} & \text{if } h_{\Sigma^u}(e) \neq u \text{ and } \psi([e]_{\Sigma^u}) \text{ is not } u\text{-extendible.} \end{cases}$$

Proof.

Claim 1 If Algorithm 3 successfully terminates then the resulting object $Pref_{\Sigma^u}$ is a branching process.

One can easily show that $Pref_{\Sigma^u}$ is an occurrence net and its labelling is a homomorphism from it to Σ^u.

Claim 2 If Algorithm 3 successfully terminates and e is an event of the resulting branching process $Pref_{\Sigma^u}$ designated cut-off by the algorithm with a corresponding configuration C then e is causally maximal, $C \approx^u_{mar} [e]_{\Sigma^u}$ and $C \lhd^u [e]_{\Sigma^u}$. Moreover, $Pref_{\Sigma^u}$ cannot be extended without consuming a post-cut-off condition.

The maximality of e is trivial.

If $\psi([e]_{\Sigma^u})$ was u-extendible then $C = \underline{\varphi}(D)$ for some u-extendible configuration D of $Pref_\Sigma^\Theta$ (otherwise the algorithm would have terminated unsuccessfully). Due to the safeness of Σ and Σ^u, a place can occur at most once in any of their reachable markings, and so $Mark_{\Sigma^u}([e]_{\Sigma^u}) = \{p\} \cup Mark_\Sigma(\psi([e]_{\Sigma^u})) \setminus S$ and, since D was u-extendible and $C = \underline{\varphi}(D)$, $Mark_{\Sigma^u}(C) = \{p\} \cup Mark_\Sigma(D) \setminus S$, i.e., $C \approx^u_{mar} [e]_{\Sigma^u}$.

If $\psi([e]_{\Sigma^u})$ was not u-extendible then $Mark_{\Sigma^u}([e]_{\Sigma^u}) = Mark_\Sigma(\psi([e]_{\Sigma^u}))$ and $C = \overline{\varphi}(D)$ for some configuration D of $Pref_\Sigma^\Theta$. If D was not u-extendible in $Pref_\Sigma^\Theta$ then $Mark_{\Sigma^u}(C) = Mark_\Sigma(D)$, i.e., $C \approx^u_{mar} [e]_{\Sigma^u}$. Otherwise, $Mark_{\Sigma^u}(C) = (\{p\} \cup Mark_\Sigma(D) \setminus S) \setminus \{p\}) \cup S = Mark_\Sigma(D)$ (note that $p \notin Mark_\Sigma(D)$ since p is not a place of Σ and $S \subseteq Mark_\Sigma(D)$ due to D being u-extendible), i.e., $C \approx^u_{mar} [e]_{\Sigma^u}$.

Proposition 7 implies that $C \lhd^u [e]_{\Sigma^u}$.

Suppose that $Pref_{\Sigma^u}$ can be extended by an event e such that ${}^\bullet e$ contains no post-cut-off conditions. We consider two cases.

- If $h_{\Sigma^u}(e) \neq u$ then ${}^\bullet e$ is a co-set in $Pref_\Sigma^\Theta$ (note that $p \notin h_{\Sigma^u}(e)$) and so $Pref_\Sigma^\Theta$ could be extended by e as well, contradicting its completeness.
- If $h_{\Sigma^u}(e) = u$ then ${}^\bullet e = \{c\}$ for some instance c of p and ${}^\bullet c = \{f\}$ for some instance f of t, since ${}^\bullet p = \{t\}$. If c is not a post-cut-off condition then f is not a cut-off event due to the maximality of cut-offs, and so the algorithm would have created e.

Hence $Pref_{\Sigma^u}$ cannot be extended without consuming a post-cut-off condition.

The proof parallels that of Proposition 8, but Claims 1 and 2 there are replaced by the ones given above. □

4.3 Concurrent insertion

For clarity of presentation, the concurrent insertion $t' \xrightarrow{\|n} t''$ in the prefix is performed in two stages: first, a place insertion $t' \xrightarrow{\textcircled{n}} t''$ is done, followed by the sequential post-insertion $t' \wr p$, as explained in Section 3.3. (In practice, these two stages can easily be combined.) Furthermore, we assume that the transformation $t' \xrightarrow{\textcircled{n}} t''$ is accepted by Algorithm 1. The following algorithm, given such a place insertion, builds $Pref_{\Sigma^p}^{\Theta^p}$ from $Pref_{\Sigma}^{\Theta}$, where Θ^p is the new cutting context defined below. Intuitively, $Pref_{\Sigma^p}^{\Theta^p}$ is obtained by adding a few p-labelled conditions to $Pref_{\Sigma}^{\Theta}$, and connecting them to instances of t' and t''.

Algorithm 4 (Place insertion in the prefix).

Step 1 *If $n = 1$ then create a new p-labelled (causally minimal) condition.*

Step 2 *For each t'-labelled event e (including cut-off events), create a new p-labelled condition c and the arc (e, c).*

Step 3 *For each t''-labelled event e (including cut-off events): If $\#_{t'}[e]_{\Sigma} = 0$ then create a new arc (c, e), where c is the minimal p-labelled condition created in Step 1; else create a new arc (c, e), where c is the (unique) p-labelled condition in the postset of the (unique) maximal (w.r.t. $\prec$) t'-labelled predecessor of e.*

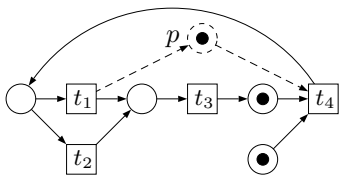
Note that there is no need for the algorithm to amend the corresponding configurations of cut-off events, as the configurations of $Pref_{\Sigma}^{\Theta}$ and $Pref_{\Sigma^p}^{\Theta^p}$ are the same.

Proposition 10 (Correctness of Algorithm 4). *Let Σ be a safe Petri net, $t' \xrightarrow{\textcircled{n}} t''$ be a place insertion accepted by Algorithm 1 and yielding a Petri net Σ^p, and $Pref_{\Sigma}^{\Theta}$ be its canonical w.r.t. the cutting context $\Theta = \left(\approx_{mar}, \lhd, \left\{\mathcal{C}_e\right\}_{e \in E_{Unf_{\Sigma}^{max}}}\right)$ prefix. Then the prefix $Pref_{\Sigma^p}$ computed by Algorithm 4 coincides with the canonical w.r.t. the cutting context $\Theta^p = \left(\approx_{mar}^{p}, \lhd, \left\{\mathcal{C}_e\right\}_{e \in E_{Unf_{\Sigma}^{max}}}\right)$ prefix of Σ^p, where $\approx_{mar}^{p}$ is the equivalence of final markings of the configurations of $Unf_{\Sigma^p}^{max}$.*

Proof. $Pref_{\Sigma^p}$ can easily be shown to be a branching process, and it has exactly the same set of events as $Pref_{\Sigma}^{\Theta}$. The key observation is that no new causal constraints are introduced by Algorithm 4, since the instances of t'' consume only the conditions produced by their causal predecessors (or the one created in Step 1 of the algorithm), and so $Pref_{\Sigma^p}$ has the same set of configurations as $Pref_{\Sigma}^{\Theta}$. Thus $\lhd$ and $\left\{\mathcal{C}_e\right\}_{e \in E_{Unf_{\Sigma}^{max}}}$ are not changed.

Let C' and C'' be two configurations o $Pref_{\Sigma}^{\Theta}$. If $C' \not\approx_{mar} C''$ then trivially $C' \not\approx_{mar}^{p} C''$, and so no new cut-off events appear in $Pref_{\Sigma^p}$. Moreover, if e is a cut-off event of $Pref_{\Sigma}^{\Theta}$ with a corresponding configuration C then $[e]_{\Sigma} \approx_{mar} C$ and the condition $Tokens([e]_{\Sigma^u}) = Tokens(C)$ (which holds due to the assumption that the transformation has been accepted by Algorithm 1) ensures that $[e]_{\Sigma^p} \approx_{mar}^{p} C$, i.e., the cut-off events of $Pref_{\Sigma}^{\Theta}$ remain cut-off events in $Pref_{\Sigma^p}^{\Theta^p}$. □

In general, the equivalence relations $\approx_{mar}$ and $\approx_{mar}^{p}$ can be different, even though their domains are the same (since $Pref_{\Sigma}^{\Theta}$ and $Pref_{\Sigma^p}^{\Theta^p}$ have the same configurations). For example, if t'' cannot fire in the future of C' and C'' then it is possible that $C' \approx_{mar} C''$ and $C' \not\approx_{mar}^{p} C''$, e.g., due to $Mark_{\Sigma^p}(C') = Mark_{\Sigma^p}(C'') \cup \{p\}$, as illustrated in the picture below: the final markings of the configurations corresponding to the executions $t_4 t_1$ and $t_4 t_2$ are equal in Σ but distinct in Σ^p.



However, $\approx_{mar}$ and $\approx^p_{mar}$ (and hence Θ and Θ^p) do coincide if t' or t'' is live, and in this case Proposition 10 can be strengthened as follows.

Proposition 11 (Strengthening of Proposition 10 in the live case). *Let Σ be a safe Petri net, $t' \xrightarrow{\textcircled{n}} t''$ be a place insertion accepted by Algorithm 1 and yielding a Petri net Σ^p, and $Pref^{\Theta}_{\Sigma}$ be its canonical w.r.t. the cutting context Θ prefix. Moreover, let t' or t'' be live. Then the prefix computed by Algorithm 4 coincides with the canonical w.r.t. Θ prefix of Σ^p.*

Proof. In the view of Proposition 10, it is enough to show that $\approx_{mar}$ and $\approx^p_{mar}$ coincide. To the contrary, suppose there are two configurations C' and C'' such that $C' \approx_{mar} C''$ but $C' \not\approx^p_{mar} C''$. Hence $Tokens(C') \neq Tokens(C'')$, and so C' and C'' satisfy the conditions of Proposition 4, which means that the transformation is not SB-preserving and thus, due to Proposition 10, should have been rejected by Algorithm 1, a contradiction. □

Now we show how to perform a concurrent insertion $t' \xrightarrow{\|n} t''$, assuming that the corresponding place insertion $t' \xrightarrow{\textcircled{n}} t''$ is accepted by Algorithm 1. The following algorithm, given such a transformation, builds $Pref^{\Theta^u}_{\Sigma^u}$ from $Pref^{\Theta}_{\Sigma}$. Intuitively, $Pref^{\Theta^u}_{\Sigma^u}$ is obtained by a successive application of Algorithms 4 and 3.

Algorithm 5 (Concurrent insertion in the prefix).

Step 1 *Perform the place insertion $t' \xrightarrow{\textcircled{n}} t''$ using Algorithm 4.*
Step 2 *Perform the sequential post-insertion $t' \wr p$ using Algorithm 3, where p is the new place created by $t' \xrightarrow{\textcircled{n}} t''$.*

Note that Algorithm 5 always terminates successfully, even though it calls Algorithm 3 (which, in general, can terminate unsuccessfully). This is because the sequential post-insertions employed by Algorithm 5 are such that Step 3 of Algorithm 3 is always successful, as $[e]_{\Sigma^p}$ is u-extendible iff C is u-extendible due to $S = \{p\}$ and ${}^\bullet p = \{t'\}$.

5 Optimisation

This section discusses several techniques allowing one to reduce the number of transformations which have to be considered, as well as to propagate information across different iterations of the algorithm for resolving encoding conflicts, avoiding thus repeating the same validity checks.

5.1 Equivalent transformations

Sometimes a sequential post-insertion $t \wr S$ yields essentially the same net as a sequential pre-insertion $S' \wr t'$, where $t \in {}^{\bullet\bullet}t'$. In such a case there is no reason to distinguish between these two transformations, e.g., one can convert the post-insertion into an equivalent pre-insertion whenever possible. Moreover, since post-insertions are always SB-preserving, there is no need to check the validity of the resulting transformation. The following proposition formalises a sufficient condition for such a conversion.

Proposition 12 (Equivalent sequential insertions). *Let $t \wr S$ be a sequential post-insertion and $S' \wr t'$ be a sequential pre-insertion. If $S \cup S' \subseteq t^\bullet \cap {}^\bullet t'$ and $|{}^\bullet p| = |p^\bullet| = 1$ for all $p \in S \cup S'$ then the resulting Petri nets are bisimular.*

5.2 Commutative transformations

Two transformations *commute* if the result of their application does not depend on the order they are applied. (Note that a transformation can become ill-defined after applying another transformation, e.g., $t \wr \{p, q\}$ becomes ill-defined after applying $t \wr p$.) One can observe that:

- a concurrent insertion always commutes with any other transition insertion;
- a sequential pre-insertion and a sequential post-insertion always commute;
- two sequential pre-insertions $S \wr t$ and $S' \wr t'$ commute iff $t \neq t'$ or $S \cap S' = \emptyset$;
- two sequential post-insertions $t \wr S$ and $t' \wr S'$ commute iff $t \neq t'$ or $S \cap S' = \emptyset$.

It is important to note that an SB-preserving transition insertion remains SB-preserving if another commuting SB-preserving transition insertion is applied first. Hence transformations whose validity has been checked can be cached, and after some transformation has been applied, the non-commuting transformations are removed from the cache and the new transformations that became possible in the modified Petri net are computed, checked for validity and added to the cache. In particular, in our application domain, there is no need to check the validity of a particular transformation if its validity was checked in some preceding iteration of the algorithm for resolving encoding conflicts.

A *composite* transition insertion is a transformation defined as the composition of a set of pairwise commutative transition insertions. Composite transformations are useful for our application domain: typically, several transitions of a new internal signal have to be inserted on each iteration of the algorithm for resolving encoding conflicts, in order to preserve the *consistency* [2, 6] of the STG, i.e., the property that for every signal s, the following two conditions hold: (i) in all executions of the STG, the first occurrence of a transition of s has the same sign (either rising of falling); (ii) the rising and falling transitions of s alternate in every trace. (Consistency is a necessary condition for implementability of an STG as a circuit.) For example, in Fig. 1(c) a composite transformation comprising two commutative sequential insertions (adding the new transitions csc^+ and csc^-) have been applied in order to resolve the encoding conflict while preserving the consistency of the STG.

Clearly, if a composite transition insertion consists of SB-preserving transition insertions then it is SB-preserving, i.e., one can freely combine SB-preserving transition insertions, as long as they are pairwise commutative.

6 Conclusions

In this paper, a method for checking correctness of transition insertions and performing them directly in the unfolding prefix is presented. The main advantage of the approach is that it avoids re-unfolding. Moreover, it yields a prefix similar to the original one, which is advantageous for visualisation and allows one to transfer some information (e.g., the yet unresolved encoding conflicts) from the original prefix to the modified one. We also demonstrated that the theory of canonical unfolding prefixes [10] is flexible enough to derive the correctness proofs for the proposed algorithms.

In future work, we intend to extend the method to other transformations, in particular concurrency reduction [5].

Acknowledgements The author would like to thank Walter Vogler for helpful comments. This research was supported by the Royal Academy of Engineering/EPSRC post-doctoral research fellowship EP/C53400X/1 (DAVAC).

References

1. R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35-8:677–691, 1986.
2. T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications.* PhD thesis, Lab. for Comp. Sci., MIT, 1987.
3. P.M. Cohn. *Universal Algebra.* Reidel, 2nd edition, 1981.

4. J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. PETRIFY: a Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, 1997.
5. J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Automatic Handshake Expansion and Reshuffling Using Concurrency Reduction. In *Proc. HWPN'98*, pages 86–110, 1998.
6. J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. *Logic Synthesis of Asynchronous Controllers and Interfaces*. Springer-Verlag, 2002.
7. J. Engelfriet. Branching Processes of Petri Nets. *Acta Informatica*, 28:575–591, 1991.
8. J. Esparza, S. Römer, and W. Vogler. An Improvement of McMillan's Unfolding Algorithm. *Formal Methods in System Design*, 20(3):285–310, 2002.
9. V. Khomenko. *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD thesis, School of Comp. Sci., Univ. of Newcastle, 2003.
10. V. Khomenko, M. Koutny, and V. Vogler. Canonical Prefixes of Petri Net Unfoldings. In *Proc. CAV'02*, Lecture Notes in Computer Science 2404, pages 582–595. Springer-Verlag, 2002. Full version: Acta Informatica 40(2):95–118, 2003.
11. V. Khomenko, M. Koutny, and A. Yakovlev. Detecting State Coding Conflicts in STG Unfoldings Using SAT. In *Proc. ICACSD'03*, pages 51–60. IEEE Computer Society Press, 2003. Full version: IOS Press, Fundamenta Informaticae, 62(2):1–21, 2004.
12. V. Khomenko, M. Koutny, and A. Yakovlev. Logic Synthesis for Asynchronous Circuits Based on Petri Net Unfoldings and Incremental SAT. In *Proc. ICACSD'04*, pages 16–25. IEEE Computer Society Press, 2004. Full version: IOS Press, Fundamenta Informaticae, 70(1–2):49–73, 2006.
13. A. Kondratyev, J. Cortadella, M. Kishinevsky, E. Pastor, O. Roig, and A. Yakovlev. Checking Signal Transition Graph Implementability by Symbolic BDD Traversal. In *Proc. DATE'1995*, pages 325–332. IEEE Computer Society Press, 1995.
14. A. Madalinski, A. Bystrov, V. Khomenko, and A. Yakovlev. Visualization and Resolution of Coding Conflicts in Asynchronous Circuit Design. In *Proc. DATE'03*, pages 926–931. IEEE Computer Society Press, 2003. Full version: IEE Proceedings: Computers & Digital Techniques 150(5):285–293, 2003.
15. T. Murata. Petri Nets: Properties, Analysis and Applications. *Proc. of the IEEE*, 77(4):541–580, 1989.
16. M. Silva, E. Teruel, and J.M. Colom. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, chapter Linear Algebraic and Linear Programming Techniques for the Analysis of Place/Transition Net Systems, pages 309–373. Springer-Verlag, 1998.
17. W. Vogler and B. Kangsah. Improved Decomposition of Signal Transition Graphs. Technical Report 2004-08, Institut für Informatik, Universität Augsburg, 2004.
18. A. Yakovlev, L. Lavagno, and A. Sangiovanni-Vincentelli. A Unified Signal Transition Graph Model for Asynchronous Control Circuit Synthesis. *Formal Methods in System Design*, 9(3):139–188, 1996.