School of Computing Science,
University of Newcastle upon Tyne

Resolution of Encoding Conflicts by Signal Insertion and Concurrency Reduction Based on STG Unfoldings

V. Khomenko, A. Madalinski and A. Yakovlev

Technical Report Series

CS-TR-858

September 2004

Published by the University of Newcastle upon Tyne,
School of Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, UK.

Resolution of Encoding Conflicts by Signal Insertion and Concurrency Reduction Based on STG Unfoldings

V. Khomenko[1], A. Madalinski[2] and A. Yakovlev[2]
[1] School of Computing Science
[2]School of Electrical, Electronic and Computer Engineering
University of Newcastle upon Tyne, NE1 7RU, UK

Abstract

A combined framework for the resolution of encoding conflicts in STG unfoldings is presented, which extends previous work by incorporating concurrency reduction in addition to signal insertion. Furthermore, a novel validity condition is proposed to justify these transformations.

1 Introduction

Signal Transition Graphs, or STGs [2,4], are widely used for specifying the behaviour of asynchronous control circuits. They are interpreted Petri nets in which transitions are labelled with the rising and falling edges of circuit signals. Synthesis based on STGs involves the following steps: (a) checking sufficient conditions for the implementability of the STG by a logic circuit; (b) modifying, if necessary, the initial STG to make it implementable; and (c) finding appropriate Boolean next-state functions for non-input signals.

A commonly used tool, PETRIFY [3, 4], performs all these steps automatically, after first constructing the reachability graph of the initial STG specification. To gain efficiency, it uses symbolic (BDD-based) techniques to represent the STG's reachable state space. While such an approach is convenient for completely automatic synthesis, it has several drawbacks: state graphs represented explicitly or in the form of BDDs are hard to visualise due to their large sizes and the tendency to obscure causal relationships and concurrency between the events, which prevents efficient interaction with the user. Moreover, the combinatorial explosion of the state space is a serious issue for highly concurrent STGs, putting practical bounds on the size of control circuits that can be synthesised. Thus PETRIFY can fail to synthesise a circuit, especially if the STG models are not constructed by a human designer but rather generated automatically from high-level hardware descriptions.

Where PETRIFY fails, other tools based on alternative techniques, and in particular those employing Petri net unfoldings, may succeed. A *finite and complete unfolding prefix* of an STG Γ is a finite acyclic net which implicitly represents all the reachable states of Γ together with transitions enabled at those states. Intuitively, it can be obtained through *unfolding* Γ, by successive firings of transition, 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 Γ has an infinite run; however, if Γ has finitely many reachable states then the unfolding eventually starts to repeat itself and can be truncated (by identifying a set of *cut-off* events) without loss of information, yielding a finite and complete prefix. Fig. 2(c) shows a finite and complete unfolding prefix (with the only cut-off event is depicted as a double box) of the STG shown in Fig. 2(a).

Efficient algorithms exist for building such prefixes [8, 9], which ensure that the number of non-cut-off events in a complete prefix can never exceed the number of reachable states of Γ. However, 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} nodes, whereas the complete prefix will coincide with the net itself.

In [9, 11] the unfolding technique was applied to the implementability analysis in step (a), viz. checking the Complete State Coding (CSC) condition [4], which requires detecting encoding conflicts between reachable states of an STG. Since STGs usually exhibit a lot of concurrency, but have rather few choice points, their unfolding prefixes are often exponentially smaller than the corresponding state graphs; in fact, in many of the experiments conducted

in [9,11] 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.

In [15] the unfolding technique was applied to step (b), in particular for enforcing the CSC condition (i.e., for the resolution of CSC conflicts), which is a necessary condition for the implementability of an STG as a circuit. A CSC conflict arises when semantically different reachable states of an STG have the same binary encoding. Fig. 2(b) shows the state graph of the STG in Fig. 2(a) with a CSC conflict between states M_1 and M_2. To resolve a CSC conflict, new signals helping to distinguish between the involved states are inserted into the specification in such a way that its 'external' behaviour does not change. (Intuitively, insertion of signals introduces additional memory into the circuit, helping to trace the current state.) In [15] a framework was developed for an interactive refinement process based on visualisation of conflict *cores*, i.e., sets of events causing encoding conflicts, which are represented at the level of finite and complete prefixes of STG unfoldings.

The work in [12] addresses step (c), where unfoldings techniques are used to derive equations for logic gates of the circuit. Together with [9, 11, 15] they form a complete design flow for complex gate synthesis of asynchronous circuits based on STG unfoldings rather than state graphs.

This paper extends the framework for the visualisation and resolution of encoding conflicts in [15] (step (b)) by incorporating the concurrency reduction transformation (which can eliminate encoding conflicts by removing some of the STG's reachable states) in addition to signal insertion.

The common belief that concurrency is crucial for performance is questionable. In a highly concurrent specification, almost all combinations of signal values are reachable, and thus Boolean minimisers cannot efficiently exploit the 'don't care' values, which results in large and slow gates in the final implementation. Moreover, transitions of the newly inserted signals delay output transitions, and hence can also increase the delay of the final circuit. Concurrency reduction can increase the number of unreachable states, thus providing more 'don't cares' for logic optimisation. Furthermore, if an encoding conflict is solved by concurrency reduction rather than signal insertion then no additional gate is required to implement this signal. Thus, the elimination of encoding conflicts by concurrency reduction may result in a faster and smaller circuit. In general, both concurrency reduction and signal insertion are required to explore a larger solution space, and considering only one of these techniques may leave out important solutions. Existing techniques either apply concurrency reduction at the state graph level [5, 14] or are restricted to specific net classes or use local transformations [1] and thus restrict the design space.

Another important contribution of this paper is a novel notion of validity, which is used to justify STG transformations used to solve encoding conflicts. We believe it better reflects the intuition than other existing notions. However, this notion is much more general and is also of independent interest: it is formulated for labelled Petri nets (of which STGs being a special case) and arbitrary transformations preserving the alphabet of the system. For example, it can be applied to justifying the concurrency increasing transformation used in [16] to convert speed-independent circuit into delay-insensitive ones.

2 Basic notation

In this section, we first present basic definitions concerning Petri nets and STGs, and then address several key concepts related to net unfoldings [4, 7–9, 17].

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* (circles) and *transitions* (boxes), collectively known as *nodes*, and $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation* (arcs). 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 ${}^\bullet Z \stackrel{\text{df}}{=} \bigcup_{z \in Z} {}^\bullet z$ and $Z^\bullet \stackrel{\text{df}}{=} \bigcup_{z \in Z} z^\bullet$, for all $Z \subseteq P \cup T$. We assume that ${}^\bullet t \neq \emptyset$, for every $t \in T$. A *marking* (tokens) of N is a multiset M of places, i.e., $M : P \to \mathbb{N} \stackrel{\text{df}}{=} \{0,1,2,\ldots\}$. An example of a Petri net with the initial marking $\{p_1, p_2\}$ is shown in Fig. 1(a).

A *net system* is a pair $\Sigma \stackrel{\text{df}}{=} (N, M_0)$ comprising a finite net $N = (P,T,F)$ and an (initial) marking M_0. A transition $t \in T$ is *enabled* at a marking M, denoted $M[t\rangle$, if for every $s \in {}^\bullet t$, $M(s) \geq 1$. Such a transition can be *executed* or *fired*, leading to the marking M' defined by $M' \stackrel{\text{df}}{=} M - {}^\bullet t + t^\bullet$, where '$-$' and '$+$' stand for the multiset difference and sum respectively. We denote this by $M[t\rangle M'$. For example, in the net system shown in Fig. 1(a) transition t_1 can fire consuming a token from p_1 and producing a token in p_3, which can be expressed as $\{p_1, p_2\}[t_1\rangle\{p_2, p_3\}$. The set of *reachable* markings of Σ is the smallest (w.r.t. $\subset$) set $[M_0\rangle$ containing M_0 and such that if $M \in [M_0\rangle$ and $M[t\rangle M'$ for some $t \in T$ then $M' \in [M_0\rangle$. For a finite sequence of transitions $\sigma = \sigma_1 \ldots \sigma_k$, we write $M[\sigma\rangle M'$ if there are markings $M_0, \ldots, M_k$ such that $M_0 = M$, $M_k = M'$, and $M_{i-1}[\sigma_i\rangle M_i$, for $i = 1, \ldots, k$.

A net system Σ is *k-bounded* if, for every reachable marking M and every place $p \in P$, $M(p) \leq k$, and *safe* if it is 1-bounded. Moreover, Σ is *bounded* if it is k-bounded for some $k \in \mathbb{N}$. One can show that the set $[M_0\rangle$ is finite iff Σ is bounded.

A transition $t \in T$ is *auto-concurrent* if there is a reachable marking M such that for every $p \in {}^\bullet t$, $M(p) \geq 2$. A

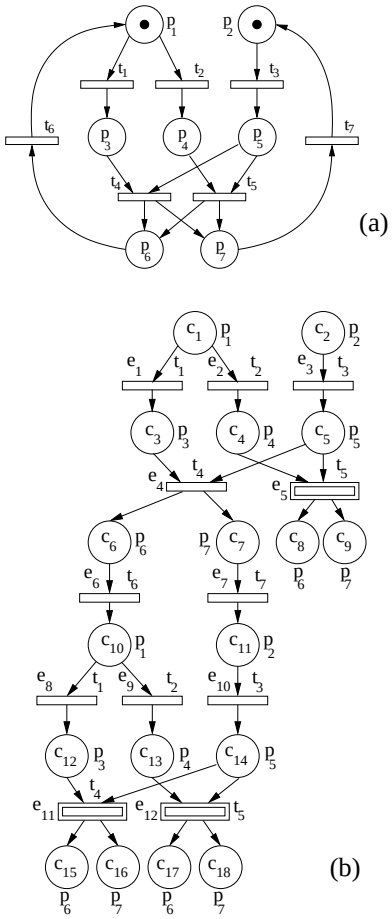


Figure 1. A Petri net (a) and one of finite and complete prefixes of its unfolding (b).

net system Σ is non-auto-concurrent if no its transition is auto-concurrent.

2.2 Branching processes and configurations

Two nodes of a net $N = (P,T,F)$, y and y', are in *structural conflict*, denoted $y\#y'$, 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 *structural 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), E is the set of *events* (transitions) and G is a *flow relation*. It is assumed that: 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 irreflexive transitive closure of G. $Min(ON)$ will denote the minimal w.r.t. $\prec$ elements of B. The relation $\prec$ is the *causality relation*. Two distinct nodes are *concurrent*, denoted $y \; co \; y'$, if neither $y\#y'$ nor $y \preceq y'$ nor $y' \preceq y$. Fig. 1(b) shows an example of an occurrence net where, e.g., the following relationships hold: $e_1 \prec e_6$, $e_4\#e_5$ (due to the choice at c_1) and $e_6 \; co \; e_7$.

A *homomorphism* from an occurrence net ON to a net system Σ is a mapping $h : B \cup E \to P \cup T$ such that: $h(B) \subseteq P$ and $h(E) \subseteq T$ (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_0 (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). In Fig. 1(b) this homomorphism is shown as labels of the nodes.

A *branching process* of Σ is a quadruple $\beta \stackrel{\text{df}}{=} (B,E,G,h)$ such that (B,E,G) is an occurrence net and h is a homomorphism from it to Σ. A branching process $\beta' = (B',E',G',h')$ of Σ is a *prefix* of a branching process $\beta = (B,E,G,h)$ of Σ, denoted $\beta' \sqsubseteq \beta$, if (B',E',G') is a subnet of (B,E,G) such that: if $e \in E'$ and $(b,e) \in G$ or $(e,b) \in G$ then $b \in B'$; if $b \in B'$ and $(e,b) \in G$ then $e \in E'$; and h' is the restriction of h to $B' \cup E'$. For each Σ there exists a unique (up to isomorphism) maximal (w.r.t. $\sqsubseteq$) branching process, called the *unfolding* of Σ (it is infinite whenever Σ has an infinite execution).

If a branching process π is such that for every its condition $b \in B$, $|b^\bullet| \leq 1$, then π is called a *process*. Since one of the transformations we are discussing is concurrency reduction, it is convenient to use a partial order rather than interleaving semantics, and our discussion will be based on *processes*, which are a partial order analog of traces. The main difference between the processes and traces is that in the former the events are ordered only partially, and thus one process can correspond to several traces, which can be obtained from it as the linearisations of the corresponding partial order. A Petri net generates a set of processes much like it generates a language.

A process can be represented as a (perhaps infinite) labelled acyclic net, with places having at most one incoming and one outgoing arc. (And a branching process can be considered as overlayed processes.) A processes is *maximal* if it is maximal w.r.t. $\sqsubseteq$, i.e., if it cannot be extended by new events. A maximal process is either infinite (though not every infinite process is maximal) or leads to a deadlock.

If π is a process and $E' \subseteq E$ is a set of events of the unfolding not belonging to π such that the events from π and E' together with their incident conditions induce a process, then this process will be denoted by $\pi \oplus E'$. Moreover, if π is finite and $U \subseteq T$, $\#_U\pi$ will denote the number of events of π with labels in U; furthermore, if $t \in T$ then $\#_t\pi \stackrel{\text{df}}{=} \#_{\{t\}}\pi$.

A *configuration* of an occurrence net is a finite set of events $C \subseteq E$ 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 $e \in E$, the configuration $[e] \stackrel{\text{df}}{=} \{f \mid f \preceq e\}$ is called the *local con-*

figuration of e. Moreover, for a set of events $E' \subseteq E$, $[E'] \stackrel{\text{df}}{=} \bigcup_{e \in E'} [e]$. Note that $[E']$ is finite whenever E' is, and $[E']$ a configuration if it is finite and no two events in E' are in structural conflict. The set of *triggers* of an event $e \in E$ is defined as $trg(e) \stackrel{\text{df}}{=} \max_{\prec}([e] \setminus \{e\})$. For example, in the net shown in Fig. 1(b) $\{e_1, e_3, e_4\}$ is a configuration whereas $\{e_1, e_2, e_3\}$ and $\{e_4, e_7\}$ are not (the former includes events in structural conflict, $e_1 \# e_2$, while the latter does not include $e_1 \prec e_4$), $[e_9] = \{e_1, e_3, e_4, e_6, e_9\}$, $[e_6, e_7] = \{e_1, e_3, e_4, e_6, e_7\}$ and $trg(e_4) = \{e_1, e_3\}$. Intuitively, a configuration is a partial-order execution, i.e., an execution where the order of firing of some of its transitions is not important; e.g., the configuration $\{e_1, e_3, e_4\}$ corresponds to two totally ordered executions: $e_1 e_3 e_4$ and $e_3 e_1 e_4$. Configurations are somewhat similar to finite processes, the difference being that the former are sets of events of the unfolding while the latter are nets. However, it is sometimes convenient to interpret a process as the set of its events, e.g., in order to apply set operations (such as $\subseteq$, $\supseteq$ and $\setminus$) to it. For example, we will denote by $\pi \setminus \pi'$ the set of events which are in π but not in π', and write $E' \subseteq \pi$ to denote the fact that every event in $E' \subseteq E$ is in π. Similarly, $\max_{\prec} \pi$ will denote the set of causally maximal events of π.

A *cut* is a maximal (w.r.t. $\subset$) set of conditions $B' \subseteq B$ such that $b \; co \; b'$, for all distinct $b, b' \in B'$. Every marking in a branching process reachable from $Min(ON)$ is a cut. Let C be a finite configuration of a branching process β. Then $Cut(C) \stackrel{\text{df}}{=} (Min(ON) \cup C^\bullet) \setminus {}^\bullet C$ is a cut; moreover, the multiset $Mark(C) \stackrel{\text{df}}{=} h(Cut(C))$ of places is a reachable marking of Σ. A marking M of Σ is *represented* in a branching process β if the latter contains a configuration C such that $M = Mark(C)$. Every marking represented in β is reachable, and every reachable marking is represented in the unfolding of Σ. For example, the cut corresponding to the configuration $\{e_1, e_3, e_4\}$ is $\{c_6, c_7\}$, and the corresponding reachable marking of Σ is $\{p_6, p_7\}$. Similar notation will be used for finite processes, e.g., $Cut(\pi)$ and $Mark(\pi)$ are defined as $Cut(C)$ and $Mark(C)$, respectively, where C comprises the events in π.

A branching process $\beta = (B, E, G, h)$ of Σ is *complete* if there is a set $E_{cut} \subseteq E$ of *cut-off* events such that, for every reachable marking M of Σ, there exists a finite configuration C of β such that $C \cap E_{cut} = \emptyset$ and $M = Mark(C)$, and for each such C and every transition t enabled by M, there is an event $e \notin C$ in β such that $h(e) = t$ and $C \cup \{e\}$ is a configuration (e may be in E_{cut}). For example, the branching process shown in Fig. 1(b) is complete w.r.t. the set $E_{cut} = \{e_5, e_{11}, e_{12}\}$ (cut-off events are shown as double boxes).

Although, in general, an unfolding can be infinite, for every bounded net system Σ one can construct a finite complete prefix of the unfolding of Σ, by choosing an appropriate set E_{cut} of cut-off events, beyond which the unfolding is not generated.

2.3 Labelled Petri nets

Definition 1 (LPN). A *labelled Petri net (LPN)* is a tuple $\Upsilon \stackrel{\text{df}}{=} (\Sigma, I, O, \ell)$, where Σ is a Petri net, $I \cap O = \emptyset$ are respectively finite sets of *inputs* (controlled by the environment) and outputs (controlled by the system), and $\ell : T \to I \cup O \cup \{\tau\}$ is a labelling function, where $\tau \notin I \cup O$ is a *silent action* (e.g., internal signals in an STG). $\Diamond$

In this notion, τ's denote internal signal transitions which are controlled by the system and not observable by the environment. They should be distinguished from 'dummy' transitions in STGs, which do not correspond to any actual signals of the circuit and are a syntactic feature. In figures, we will denote inputs by i or i_k, and outputs by o or o_k.

A *branching process* of an LPN $\Upsilon = (\Sigma, I, O, \ell)$ is a branching process of Σ augmented with an additional labelling of its events, $(\ell \circ h) : E \to I \cup O \cup \{\tau\}$, and processes of an LPN are defined in a similar way. If $\pi = (B, E, G, h)$ is a process of an LPN $\Upsilon = (\Sigma, I, O, \ell)$ then the *abstraction of π w.r.t.* ℓ is the labelled partially-ordered set (with the labels in $I \cup O$) $\text{abs}_\ell(\pi) \stackrel{\text{df}}{=} (E', \prec', \ell')$ where: $E' = \{e \in E \mid \ell(h(e)) \neq \tau\}$; $\prec'$ is the restriction of $\prec$ to $E' \times E'$; and $\ell' : E' \to I \cup O$ is such that for all $e \in E'$, $\ell'(e) = \ell(h(e))$. We will write $\text{abs}(\pi)$ instead of $\text{abs}_\ell(\pi)$ if ℓ is obvious from the context.

An LPN is *input-proper* if no input event in its unfolding is triggered by an internal event, i.e., if for every event e in the unfolding such that $\ell(h(e)) \in I$, and for every event $f \in trg(e)$, $\ell(h(f)) \neq \tau$.

2.4 Signal Transition Graphs

A *Signal Transition Graph (STG)* is a triple $\Gamma \stackrel{\text{df}}{=} (\Sigma, Z, \lambda)$ such that $\Sigma = (N, M_0)$ is a net system, Z is a finite set of signals generating a finite alphabet $Z^\pm \stackrel{\text{df}}{=} Z \times \{+, -\}$ of *signal transition labels*, and $\lambda : T \to Z^\pm$ is a labelling function. The signal transition labels are of the form z^+ or z^-, and denote the transitions of a signal $z \in Z$ from 0 to 1 (rising edge), or from 1 to 0 (falling edge), respectively. We will also denote by $z^\pm$ a transition of signal z if its direction is not particularly important. For the graphical representation of STGs a short-hand notation is used, where a transition can be connected directly to another transition if the place 'in the middle of the arc' has one incoming and one outgoing arc. An example of an STG specification of a VME bus controller regulating the communication between a device and a bus through a data transceiver is shown in Fig. 2(a).

We associate with the initial marking of Γ a binary vector $v^0 \stackrel{\text{df}}{=} (v_1^0, \ldots, v_{|Z|}^0) \in \{0,1\}^{|Z|}$, where v_i^0 corresponds to the initial value of signal $z_i \in Z$. Moreover, with a sequence of transitions σ we associate an integer *signal change vector*

$\nu^\sigma \stackrel{\text{df}}{=} (\nu^\sigma_1, \nu^\sigma_2, \ldots, \nu^\sigma_{|Z|}) \in \mathbb{Z}^{|Z|}$, so that each ν^σ_i is the difference between the numbers of the occurrences of z_i^+– and z_i^-–labelled transitions in σ.

Γ is *consistent* if, for every reachable marking M, all firing sequences σ from M_0 to M have the same *encoding* $Code(M) \stackrel{\text{df}}{=} \nu^0 + \nu^\sigma$, and this vector is binary, i.e., $Code(M) \in \{0,1\}^{|Z|}$. Such a property guarantees that, for every signal $z \in Z$, the STG satisfies the following two properties: (i) the first occurrence of z in the labelling of any firing sequence of Γ starting from M_0 has the same sign (either rising of falling); and (ii) the rising and falling labels of z alternate in any firing sequence of Γ. All STGs considered in the sequel are assumed to be consistent.

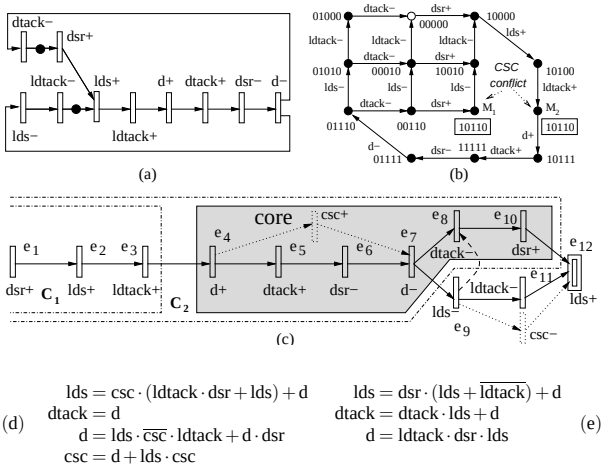
The *state graph* of a consistent STG Γ is a tuple $SG_\Gamma \stackrel{\text{df}}{=} (S, A, s_0, Code)$ such that: $S \stackrel{\text{df}}{=} [M_0\rangle$ is the set of *states*; $A \stackrel{\text{df}}{=} \{M \xrightarrow{\lambda(t)} M' \mid M \in [M_0\rangle \wedge M[t\rangle M'\}$ is the set of *transitions*; $s_0 \stackrel{\text{df}}{=} M_0$ is the *initial state*; and $Code : S \to \{0,1\}^{|Z|}$ is the *state assignment* function, as defined above for markings. Fig. 2(b) shows the state graph of the STG depicted in part (a) of this figure together with the encodings of all the reachable states.

Signals in Z are partitioned into input signals, Z_I, output signals, Z_O, and internal signals, Z_τ. Input signals are assumed to be generated by the environment, while *local* (i.e., output and internal) signals are produced by the logical gates of the circuit. Logic synthesis derives a Boolean function $F_z(z_1, \ldots, z_{|Z|})$ for each signal $z \in Z_O \cup Z_\tau$, which requires the conditions for the enabling of each output signal transition to be determined without ambiguity by the encoding of each reachable state. To capture this, let $Loc(M) \stackrel{\text{df}}{=} \{z \in Z_O \cup Z_\tau \mid \exists t \in T : M[t\rangle \wedge \lambda(t) = z^\pm\}$ be the set of local signals enabled at state M. Two states of SG_Γ are in *CSC conflict* if they have the same encoding but different sets of enabled local signals. Γ satisfies the *Complete State Coding (CSC)* property if no two states of SG_Γ are in CSC conflict. Fig. 2(b) illustrates a CSC conflict between two different markings, M_1 and M_2, that have the same encoding, 10110, but $Loc(M_1) = \{lds\} \neq Loc(M_2) = \{d\}$. This means that, e.g., the value of $F_{lds}(1,0,1,1,0)$ is ill-defined (it should be 0 according to M_1 and 1 according to M_2), and thus *lds* is not implementable as a logic gate. To cope with this, the STG should be transformed, e.g., by adding new internal signals, so that the resulting STG satisfies the CSC property.

Note that an STG Γ can be considered as a special case of an LPN with the same underlying Petri net, $I \stackrel{\text{df}}{=} Z_I$, $O \stackrel{\text{df}}{=} Z_O$ and ℓ defined as

$$\ell(t) \stackrel{\text{df}}{=} \begin{cases} \tau & \text{if } \lambda(t) = z^\pm \wedge z \in Z_\tau \\ z & \text{if } \lambda(t) = z^\pm \wedge z \notin Z_\tau \,. \end{cases}$$

A *branching process* of an STG $\Gamma = (\Sigma, Z, \lambda)$ is a branching process of Σ augmented with an additional labelling of its events, $(\lambda \circ h) : E \to Z^\pm$. One can easily check the consistency of Γ once its finite and complete prefix has been built [17]. A complete unfolding prefix of the STG shown in Fig. 2(a) is shown in part (c) of this figure, where e_{12} is a cut-off event.



inputs: *dsr*, *ldtack*; **outputs:** *lds*, *d*, *dtack*; **internal:** *csc*

Figure 2. VME bus controller: the STG for the read cycle (a), its state graph showing a CSC conflict (b), its unfolding prefix with the corresponding conflict core (c), and the equations for signal insertion (d) and concurrency reduction (e). The signal order in binary encodings is: *dsr, dtack, lds, ldtack, d***.**

3 Valid transformations

The notion of validity for signal insertion is quite easy — one can justify such a transformation in terms of weak bisimulation, which is well-studied. For a concurrency reduction (or transformations in general), the situation is more difficult: the original and transformed systems are typically not even language-equivalent; deadlocks can disappear (e.g., the deadlocks in Dining Philosophers can be eliminated by fixing the order in which forks are taken); deadlocks can be introduced; transitions can become dead; even the language inclusion may not hold (some transformations, e.g., converting a speed-independent circuit into a delay-insensitive one [16] can increase the concurrency of inputs, which in turn *extends* the language). For the sake of generality, we discuss arbitrary transformations (not necessarily concurrency reductions or signal insertions).

Intuitively, there are four aspects to a valid transformation:

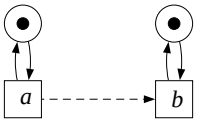
I/O interface preservation The transformation must preserve the interface between the circuit and the environment. In particular, no input transition can be 'delayed' by newly inserted signals or ordering constraints.

Conformation Bounds the behaviour from above, i.e., requires that the transformation introduces no 'wrong' behaviour. Note that certain extensions of behaviour are valid, e.g., two inputs in sequence may be accepted concurrently [6, 16], extending the language.

Liveness Bounds the behaviour from below, i.e., requires that no 'interesting' behaviour is completely eliminated by the transformation.

Technical restrictions It might happen that a valid transformation is still unacceptable because the STG becomes unimplementable or because of some other technical restriction. For example, one usually requires the transformation to preserve the speed-independence and boundedness of the STG [4, 5].

In the example below, the original LPN is bounded (in fact, safe), whereas the concurrency reduction shown by the dashed arc yields an unbounded LPN, even though its behaviour may be valid.



In this section we discuss in the described framework the notions of validity proposed in [5, 6] and present a new one, which, in our opinion, better reflects the intuition of what a valid transformation is. Since the first and the last aspects are well-studied [4], we will concentrate on the remaining two aspects, viz. conformation and liveness.

3.1 Critical overview of previous validity notions

The liveness restrictions imposed on transformations in [5] require that (i) no events become dead, and (ii) no (new) deadlock states appear. As the example in Fig. 3 shows, these restrictions are not sufficient to guarantee the correctness of the modified LPN. Indeed, the enabling region of output *o* has not become empty, and the set of deadlocks has not changed, even though the transformation is clearly invalid: in the original specification, output *o* is always produced, whereas in the transformed one the environment can prevent *o* from occurring by repeatedly choosing i_1 rather then i_2.

In [5] a notion of *conformation* is introduced. However, it cannot express the liveness conditions, e.g., the Universal Do-Nothing module, accepting all inputs but not producing any outputs, conforms to any specification with the same alphabet; thus *one cannot require the circuit to do anything.* The other notion introduced in [5] is based on the existence of a winning strategy in a certain infinite game, and is quite complicated. In this paper we propose an elegant bisimulation-style notion which takes the liveness into account.

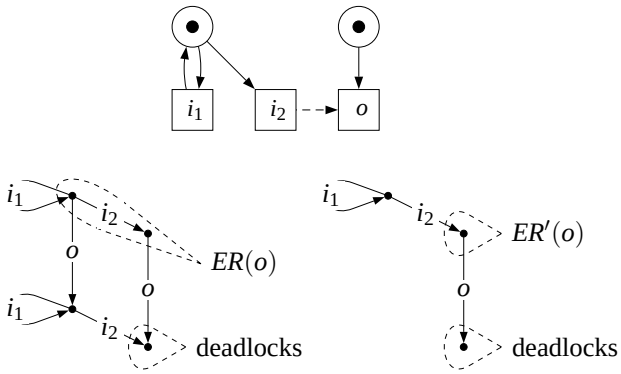


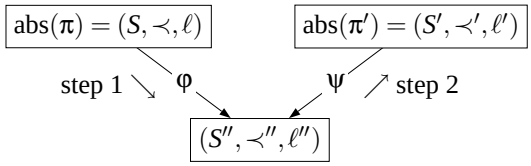
Figure 3. Liveness problem: an LPN with the concurrency reduction shown by the dashed arc together with its state graphs before and after the transformation.

3.2 Our notion of validity

For the sake of generality, we discuss arbitrary LPNs (STGs being a special kind of them). We assume that the transformation does not change the inputs and outputs of the system, and we will denote by Υ and Υ' the original and transformed LPNs, respectively. Since one of the transformations we are discussing is concurrency reduction, it is convenient to use a partial order rather than interleaving semantics, and our discussion will be based on processes of LPNs.

Given processes π of Υ and π' of Υ', we define a relation between their abstractions, $\mathrm{abs}(\pi)$ and $\mathrm{abs}(\pi')$, which holds iff in π' the inputs are no less concurrent and the outputs are no more concurrent than in π. That is, the transformation is allowed, on one hand, to relax the assumptions about the order in which the environment will produce input signals, and, on the other hand, to restrict the order in which outputs are produced. Thus the modified LPN will not produce new failures and will not cause new failures in the environment.

Intuitively, $\mathrm{abs}(\pi)$ and $\mathrm{abs}(\pi')$ are bound by this relation iff $\mathrm{abs}(\pi)$ can be transformed into $\mathrm{abs}(\pi')$ in two steps (see the picture below): (i) the ordering constraints for inputs are relaxed (yielding a new order $\prec''$, which is a relaxation of $\prec$); (ii) new ordering constraints for outputs are added, yielding $\mathrm{abs}(\pi')$ (thus, $\prec''$ is also a relaxation of $\prec'$).

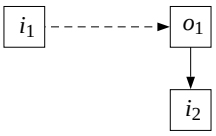


Below we give two alternative definitions of such a relation.

Definition 2. Let π and π' be processes of Υ and Υ', respectively, $\mathrm{abs}(\pi) = (S, \prec, \ell)$ and $\mathrm{abs}(\pi') = (S', \prec', \ell')$. We define $\mathrm{abs}(\pi) \bowtie^{*} \mathrm{abs}(\pi')$ if there exist a labelled partially ordered set $(S'', \prec'', \ell'')$ and one-to-one mappings $\varphi : \mathrm{abs}(\pi) \to (S'', \prec'', \ell'')$ and $\psi : \mathrm{abs}(\pi') \to (S'', \prec'', \ell'')$ preserving the labels and such that:

- $\prec'' = \varphi(\prec) \cap \psi(\prec')$ ($\prec''$ is a relaxation of $\prec$ and $\prec'$);
- if e is an output event and $f \prec e$ then $\varphi(f) \prec'' \varphi(e)$ (in step 1, existing ordering constraints for outputs are preserved);
- if e' is an input event and $f' \prec' e'$ then $\psi(f') \prec'' \psi(e')$ (in step 2, no new ordering constraints for inputs can appear). $\Diamond$

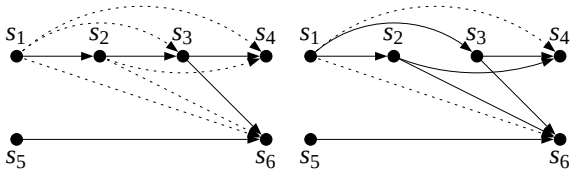
This definition turns out to be too restrictive in practice, e.g., in the picture below, the abstraction of the process obtained by adding the dashed arc is not bound to the abstraction of the original one by $\bowtie^{*}$, since delaying o indirectly delays i_2.



In practice, one often can assume the *weak fairness*, i.e., that a transition cannot remain enabled forever: it must either fire or be disabled by another transition firing. Under this assumption, the transformation in the picture above is quite reasonable. The following notion is less restrictive than the one given in Definition 2. Unlike that definition, it is mostly concerned with *direct* ordering constraints.

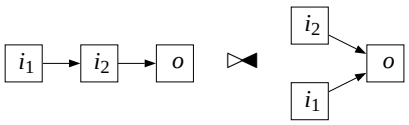
Let $(S, \prec)$ be a partially ordered set and $s \in S$. An $s' \in S$ is a *direct predecessor* of s if $s' \prec s$ and there is no $s'' \in S$ such that $s' \prec s'' \prec s$. We will denote by $DP_{\prec}(s)$ the set of direct predecessors of an $s \in S$.

Example 1. *Consider the acyclic graph representing a partial order, with the direct predecessor relation shown as solid lines and the transitive arcs shown as dotted lines. The second graph represents the relaxation of this order obtained by eliminating the arc (s_2, s_3). Note that some of the transitive arcs are now a part of the direct predecessor relation.*

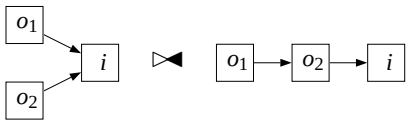


Definition 3. Let π and π' be processes of Υ and Υ', respectively, $\mathrm{abs}(\pi) = (S, \prec, \ell)$ and $\mathrm{abs}(\pi') = (S', \prec', \ell')$. We define $\mathrm{abs}(\pi) \bowtie \mathrm{abs}(\pi')$ if there exist a labelled partially ordered set $(S'', \prec'', \ell'')$ and one-to-one mappings $\varphi : \mathrm{abs}(\pi) \to (S'', \prec'', \ell'')$ and $\psi : \mathrm{abs}(\pi') \to (S'', \prec'', \ell'')$ preserving the labels and such that:

- $\prec'' = \varphi(\prec) \cap \psi(\prec')$ ($\prec''$ is a relaxation of $\prec$ and $\prec'$);
- if e is an output event and $f \in DP_{\prec}(e)$ then $\varphi(f) \in DP_{\prec''}(\varphi(e))$ (in step 1, existing *direct* ordering constraints for outputs are preserved, and existing indirect ones can become direct, e.g., as in the picture below);



- if e' is an input event and $f' \in DP_{\prec'}(e')$ then $\psi(f') \in DP_{\prec''}\psi(e')$ (in step 2, no new *direct* ordering constraints for inputs can appear, e.g., as in the picture below).



$\Diamond$

The following proposition states that $\bowtie$ is less restrictive than $\bowtie^{*}$.

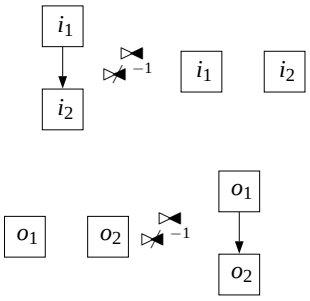
Proposition 1. *Let $(S, \prec, \ell)$ and $(S', \prec', \ell')$ be labelled partially ordered sets such that $(S, \prec, \ell) \bowtie^{*} (S', \prec', \ell')$. Then $(S, \prec, \ell) \bowtie (S', \prec', \ell')$.*

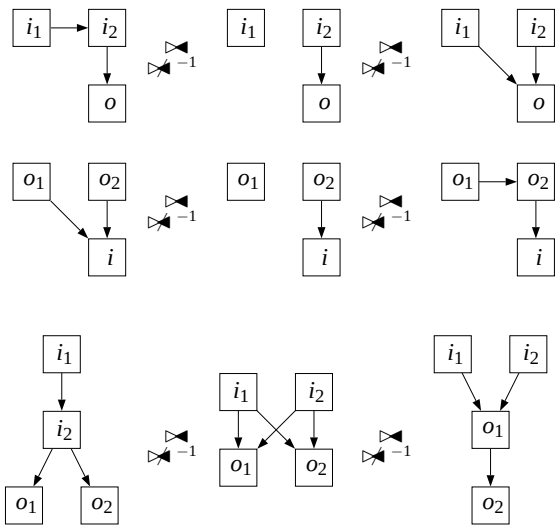
Proof. Follows from comparison of Definitions 2 and 3 taking into account the following facts:

- if $s' \in DP_{\prec}(s)$ then $s' \prec s$;
- if $\prec'$ is a relaxation of $\prec$, $s' \in DP_{\prec}(s)$ and $s' \prec' s$ then $s' \in DP_{\prec'}(s)$. $\square$

In the rest of this paper we will assume that the weak fairness condition holds and use $\bowtie$ rather than $\bowtie^{*}$.

Example 2. *The following hold:*





Note that $\rhd\!\lhd$ is an order (if we do not distinguish order-isomorphic partially ordered sets). In the sequel, slightly abusing the notation, we will write $\pi \rhd\!\lhd \pi'$ instead of $\mathrm{abs}(\pi) \rhd\!\lhd \mathrm{abs}(\pi')$.

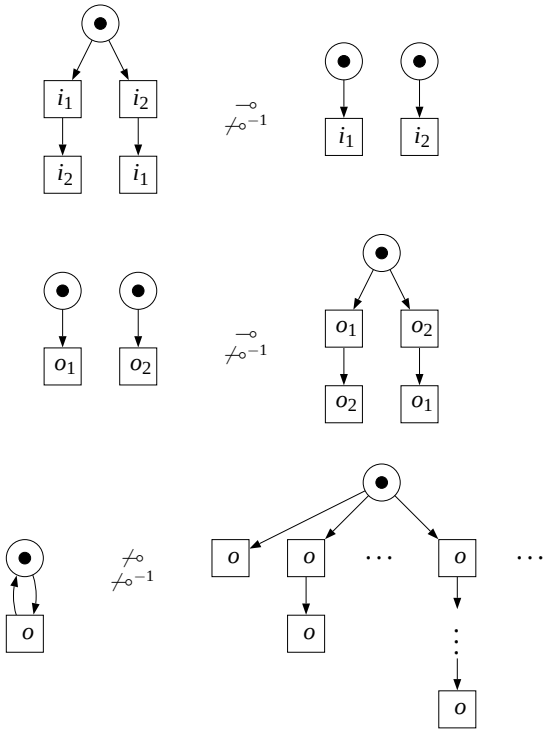
Definition 4 (Validity). Υ' is a *valid realisation* of Υ, denoted $\Upsilon \multimap \Upsilon'$, if there is a relation $\propto$ between the finite processes of Υ and Υ' such that $\pi_\emptyset \propto \pi'_\emptyset$ (where $\pi_\emptyset$ and $\pi'_\emptyset$ are the empty processes of Υ and Υ', respectively), and for all finite processes π and π' such that $\pi \propto \pi'$:

- $\pi \rhd\!\lhd \pi'$
- For all maximal processes $\Pi' \sqsupseteq \pi'$, and for all finite processes $\widehat{\pi}' \sqsupseteq \pi'$ such that $\widehat{\pi}' \sqsubseteq \Pi'$, there exist finite processes $\widetilde{\pi}' \sqsupseteq \widehat{\pi}'$ and $\widetilde{\pi} \sqsupseteq \pi$ such that $\widetilde{\pi}' \sqsubseteq \Pi'$ and $\widetilde{\pi} \propto \widetilde{\pi}'$.
- For all maximal processes $\Pi \sqsupseteq \pi$, and for all finite processes $\widehat{\pi} \sqsupseteq \pi$ such that $\widehat{\pi} \sqsubseteq \Pi$, there exist finite processes $\widetilde{\pi} \sqsupseteq \widehat{\pi}$ and $\widetilde{\pi}' \sqsupseteq \pi'$ such that $\widetilde{\pi} \sqsubseteq \Pi$ and $\widetilde{\pi} \propto \widetilde{\pi}'$. $\Diamond$

Intuitively, every activity of Υ is *eventually* performed by Υ' (up to the $\rhd\!\lhd$ relation) and cannot be pre-empted due to choices, and vice versa, i.e., Υ' and Υ simulate each other with a finite delay. Note that $\multimap$ is a pre-order, i.e., a sequence of two valid transformations is a valid transformation.

In this definition, considering maximal processes is essential. Indeed, according to this notion the transformation in Fig. 3 is not valid, since in the original LPN no extension of the process comprising an instance of o within the maximal process comprising an infinite sequence of instances of i_1 and an instance of o has a corresponding (in terms of the $\rhd\!\lhd$ relation) process in the transformed LPN, which would have to fire i_2 before it is able to fire o.

Example 3. *The following hold:*



4 Concurrency reduction

Now we give a general definition of concurrency reduction (see Fig. 4).

Definition 5 (Concurrency reduction). Given an LPN $\Upsilon = (\Sigma, I, O, \ell)$ where $\Sigma = (P, T, F, M_0)$, a non-empty set of transitions $U \subset T$, a transition $t \in T \setminus U$ and an $n \in \mathbb{N}$, the transformation $U \overset{n}{\dashrightarrow} t$, yielding an LPN $\Upsilon' = (\Sigma', I, O, \ell)$ with $\Sigma' = (P', T, F', M'_0)$ is defined as follows:

- $P' \overset{\mathrm{df}}{=} P \cup \{p\}$, where $p \notin P \cup T$ is a new place;
- $F' \overset{\mathrm{df}}{=} F \cup \{(u,p) | u \in U\} \cup \{(p,t)\}$;
- For all places $q \in P$, $M'_0(q) \overset{\mathrm{df}}{=} M_0(q)$, and $M'_0(p) \overset{\mathrm{df}}{=} n$.

We will write $U \dashrightarrow t$ instead of $U \overset{0}{\dashrightarrow} t$ and $u \overset{n}{\dashrightarrow} t$ instead of $\{u\} \overset{n}{\dashrightarrow} t$. $\Diamond$

Note that concurrency reduction cannot add new behaviour to the system — it can only restrict it. Furthermore, one can easily show that if a concurrency reduction $U \overset{n}{\dashrightarrow} t$ such that $\ell(t) \notin I$ is applied to an input-proper LPN Υ, then the resulting LPN Υ' is also input proper.

The proposition below is quite technical. Its advantage is that the conditions are imposed only on the *original* LPN. Some of these conditions are simplified later for special net classes.

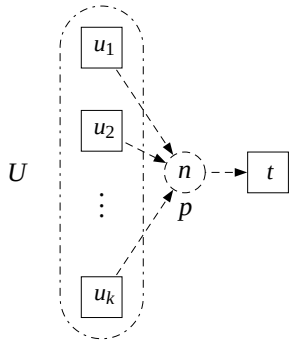


Figure 4. Concurrency reduction $U \xrightarrow{n} t$.

Proposition 2 (Validity condition for a concurrency reduction). *Let $U \xrightarrow{n} t$ be a concurrency reduction transforming an input-proper LPN $\Upsilon = (\Sigma, I, O, \ell)$ into $\Upsilon' = (\Sigma', I, O, \ell)$, such that $\ell(t) \notin I$ and for each finite process π of Υ, each t-labelled event e and each maximal process $\Pi \sqsupseteq \pi \oplus \{e\}$ of Υ, there exists a finite process $\widetilde{\pi} \sqsubseteq \Pi$ of Υ such that $\widetilde{\pi} \sqsupseteq \pi \oplus \{e\}$, $e \in \max_{\prec} \widetilde{\pi}$, $t \notin h(\widetilde{\pi} \setminus (\pi \oplus \{e\}))$ and $n + \#_U \widetilde{\pi} \geq \#_t \widetilde{\pi}$. Then $\Upsilon \multimap \Upsilon'$.*

Proof. We define the relation $\propto$ between the finite processes of Υ and Υ' as follows: $\pi \propto \pi'$ iff there exists a one-to-one mapping ξ between the nodes of π and non-p-labelled nodes of π', where p is the place added by the concurrency reduction (note that π does not contain p-labelled conditions), such that for every condition c and event e of π:

- $h(c) = h(\xi(c))$ and $h(e) = h(\xi(e))$ (i.e., ξ preserves the labels of events and conditions).
- $e \in {}^\bullet c$ iff $\xi(e) \in {}^\bullet \xi(c)$ and $e \in c^\bullet$ iff $\xi(e) \in (\xi(c))^\bullet$ (i.e., ξ preserves the environments of the conditions; note that the environments of the events in π' may contain additional p-labelled conditions which are not present in π).

Intuitively, $\pi \propto \pi'$ iff π' can be obtained from π by adding a few p-labelled conditions and the corresponding arcs. Note that according to this definition, $\pi_\emptyset \propto \pi'_\emptyset$. We proceed by proving a few properties of this relation.

Claim 1: if $\pi \propto \pi'$ then $\pi \rhd\!\lhd \pi'$.

We choose the labelled partially ordered set $(S'', \prec'', \ell'')$ in Definition 3 to be $\mathrm{abs}(\pi)$, φ to be the identity mapping, and ψ to be ξ^{-1} restricted to the events labelled by non-internal transitions. Both φ and ψ preserve the labels and $\prec'' = \varphi(\prec) \cap \psi(\prec')$ (the latter holds because $\prec'' = \prec$ by definition, and all the arcs in π are also present in π', i.e., $\prec$ is a relaxation of $\prec'$).

Since $\prec'' = \prec$, the only property we still have to prove is that if e' is an input event and $f' \in DP_{\prec'}(e')$ then $\psi(f') \in DP_{\prec''}\psi(e')$. It holds because $\ell(t) \notin I$ and Υ is input proper, and thus no input event is delayed (either directly or via a chain of τ-labelled events) by the transformation.

Claim 2: $\xi(\mathit{Cut}(\pi)) \subseteq \mathit{Cut}(\pi')$.

By the definition of the $\propto$ relation, ξ preserves the environments of the conditions, and the claim easily follows from the fact that a condition in a finite process belongs to the cut iff its postset is empty. Note that in general $\xi(\mathit{Cut}(\pi)) \neq \mathit{Cut}(\pi')$, since the latter may contain p-labelled conditions which are not present in the former.

Claim 3: if $\pi \propto \pi'$ and π' can be extended by a finite set of events E' then π can be extended by a finite set of events E such that $h(E) = h(E')$ and $\pi \oplus E \propto \pi' \oplus E'$.

If E' is a singleton $\{e'\}$ then ${}^\bullet e' \in \mathit{Cut}(\pi')$ and the result follows from Claim 2 (even if e' is t-labelled, since there is no p-labelled place in Υ able to restrict the firing of t). Since any finite extension of π' can be obtained by a finite sequence of single-event extensions, the claim follows by induction.

Claim 4: if $\pi \propto \pi'$, π can be extended by a finite set of events E and $t \notin h(E)$ then π' can be extended by a finite set of events E' such that $h(E) = h(E')$ and $\pi \oplus E \propto \pi' \oplus \{E'\}$.

The proof is very similar to that of Claim 3 (but we do not have to take care of t-labelled events, which are the only ones consuming tokens from place p in Υ').

Claim 5: if $\pi \propto \pi'$, $p \in \mathit{Mark}(\pi')$ and π can be extended by a t-labelled event e then π' can be extended by a t-labelled event e' in such a way that $\pi \oplus \{e\} \propto \pi' \oplus \{e'\}$.

Since ${}^\bullet e \in \mathit{Cut}(\pi)$, the result follows from Claim 2 and the presence of a p-labelled condition in $\mathit{Cut}(\pi')$. Note that e' can be chosen in a non-unique way if there are several p-labelled conditions in $\mathit{Cut}(\pi')$.

Now we need to demonstrate that the relation $\propto$ satisfies Definition 4, i.e., assuming that $\pi \propto \pi'$ we need to show that

1. $\pi \rhd\!\lhd \pi'$.

This property holds by Claim 1.

2. For all maximal processes $\Pi' \sqsupseteq \pi'$, and for all finite processes $\widehat{\pi}' \sqsupseteq \pi'$ such that $\widehat{\pi}' \sqsubseteq \Pi'$, there exist finite processes $\widetilde{\pi}' \sqsupseteq \widehat{\pi}'$ and $\widetilde{\pi} \sqsupseteq \pi$ such that $\widetilde{\pi}' \sqsubseteq \Pi'$ and $\widetilde{\pi} \propto \widetilde{\pi}'$.

This property follows from Claim 3.

3. For all maximal processes $\Pi \sqsupseteq \pi$, and for all finite processes $\widehat{\pi} \sqsupseteq \pi$ such that $\widehat{\pi} \sqsubseteq \Pi$, there exist finite processes $\widetilde{\pi} \sqsupseteq \widehat{\pi}$ and $\widetilde{\pi}' \sqsupseteq \pi'$ such that $\widetilde{\pi} \sqsubseteq \Pi$ and $\widetilde{\pi} \propto \widetilde{\pi}'$.

Since any extension $\widehat{\pi} \sqsupseteq \pi$ can be obtained by a sequence of single-event extensions, it suffices to prove this property for the case $\widehat{\pi} = \pi \oplus \{e\} \sqsubseteq \Pi$.

If $h(e) \neq t$ then the result follows from Claim 4. If $h(e)=t$ then there exists a finite $\widetilde{\pi} \sqsubseteq \Pi$ such that $\widetilde{\pi} \sqsupseteq \pi \oplus \{e\}$, $e \in \max_{\prec} \widetilde{\pi}$, $t \notin h(\widetilde{\pi} \setminus (\pi \oplus \{e\}))$ and $n + \#_U \widetilde{\pi} \geq \#_t \widetilde{\pi}$. Since e is a maximal event of both $\pi \oplus \{e\}$ and $\widetilde{\pi}$, the events in $E \stackrel{\text{df}}{=} \widetilde{\pi} \setminus (\pi \oplus \{e\})$ are concurrent to e. Moreover, $t \notin h(E)$, and so by Claim 4, $\pi \oplus E \propto \pi' \oplus E'$ and $h(E) = h'(E')$ for some E'.

Since $\widetilde{\pi} = (\pi \oplus E) \oplus \{e\}$, e is t-labelled and $t \notin U$, $\#_U \widetilde{\pi} = \#_U((\pi \oplus E) \oplus \{e\}) = \#_U(\pi \oplus E)$ and $\#_t \widetilde{\pi} = \#_t((\pi \oplus E) \oplus \{e\}) = \#_t(\pi \oplus E) + 1$, and thus $n + \#_U \widetilde{\pi} \geq \#_t \widetilde{\pi}$ implies $n + \#_U(\pi \oplus E) \geq \#_t(\pi \oplus E) + 1 > \#_t(\pi \oplus E)$.

Since $h(\pi) = h'(\pi')$ and $h(E) = h'(E')$, $h(\pi \oplus E) = h'(\pi' \oplus E')$, and thus $n + \#_U(\pi' \oplus E') > \#_t(\pi' \oplus E')$, i.e., $p \in \mathit{Mark}(\pi' \oplus E')$, and so by Claim 5, $\pi' \oplus E'$ can be extended by a t-labelled event e' in such a way that $\pi \oplus \{e\} \sqsubseteq (\pi \oplus E) \oplus \{e\} = \widetilde{\pi} \propto \widetilde{\pi}' = (\pi' \oplus E') \oplus \{e'\} \sqsupseteq \pi'$. □

This validity condition for general LPNs is quite complicated. It can be somewhat simplified if t is a non-auto-concurrent transition.

Proposition 3 (Validity condition for a concurrency reduction on non-auto-concurrent nets). *Let $U \stackrel{n}{\dashrightarrow} t$ be a concurrency reduction transforming an input-proper LPN $\Upsilon = (\Sigma, I, O, \ell)$ into $\Upsilon' = (\Sigma', I, O, \ell)$, such that $\ell(t) \notin I$, and t is non-auto-concurrent and such that for each t-labelled event e and for each maximal process $\Pi \supseteq [e]$ of Υ there is a finite set $E_U \subseteq \Pi$ of events with labels in U concurrent to e such that $n + \#_U[e] + |E_U| \geq \#_t[e]$. Then $\Upsilon \multimap \Upsilon'$.*

Proof. Any maximal process $\Pi \sqsupseteq \pi \oplus \{e\}$ is also a maximal extension of $[e]$. Take $\widetilde{\pi} \stackrel{\text{df}}{=} (\pi \oplus \{e\}) \cup [E_U]$; note that $\widetilde{\pi} \sqsubseteq \Pi$ since $\pi \oplus \{e\} \sqsubseteq \Pi$ and $E_U \subseteq \Pi$. All the events in E_U are concurrent to e, and thus $e \in \max_{\prec} \widetilde{\pi}$. Since t is non-auto-concurrent, all the t-labelled events in $\widetilde{\pi}$ are in $[e]$, i.e., $t \notin h(\widetilde{\pi} \setminus (\pi \oplus \{e\}))$ and $\#_t \widetilde{\pi} = \#_t[e]$; thus, since $\#_U \widetilde{\pi} \geq \#_U[e] + |E_U|$,

$$n + \#_U \widetilde{\pi} - \#_t \widetilde{\pi} = n + \#_U \widetilde{\pi} - \#_t[e] \geq$$
$$n + \#_U[e] + |E_U| - \#_t[e] \geq 0 \,,$$

so, by Proposition 2, $\Upsilon \multimap \Upsilon'$. □

Remark 1. *This proposition requires the non-auto-concurrency of a particular transition rather than the absence of two transitions with the same label which can be executed concurrently. That is, the non-auto-concurrency is required not on the level of LPN, but rather on the level of the underlying Petri net. In particular, the non-auto-concurrency is guaranteed for safe Petri nets.*

The next section explains how concurrency reduction can be employed for resolution of encoding conflicts in STG unfoldings.

5 Resolution of encoding conflicts

At the level of unfoldings, encoding conflicts can be compactly represented using conflict *cores* [15]. Encoding conflicts can be resolved by either adding auxiliary signals or by concurrency reduction. The former approach was studied in [15], where additional signals are employed to disambiguate states having the same binary encodings. The latter makes some of the states unreachable and thus can eliminate encoding conflicts.

The resolution of encoding conflicts by signal insertion is illustrated in Fig. 2(c), where the signal *csc* is inserted concurrently to existing transitions in order to minimise the latency. The logic equations for this solution are shown in Fig. 2(d). Fig. 2(c) also shows how to reduce the concurrency between lds^- and $dtack^-$ so that state M_1 is removed from the reachability graph shown in Fig. 2(b), which in turn resolves the encoding conflict. One can see that in this example the equations for the signal insertion are more complex then those obtained by concurrency reduction. This can be explained as follows. The concurrent insertion of auxiliary transitions avoids delaying any output transitions but increases the state space and thus reduces the 'don't care' set, which is used for logic optimisation. Moreover, signal insertion increases the number of support variables for output signals. Furthermore, an additional logic gate is required to implement the inserted signal. In contrast, concurrency reduction reduces the state space, increasing the 'don't care' set, while delaying the output transition $dtack^-$, making it wait until lds^- completes.

In general, concurrency reduction produces smaller circuits, and it may also be the case that the resulting circuit is faster due to simplification of the gates. Thus, even though the system manifests less concurrency, it might be actually faster due to the events taking less time to fire. On the other hand, there are situations when signal insertion produces better solutions.

A combined framework is presented here, which uses both signal insertion and concurrency reduction to eliminate cores and the corresponding encoding conflicts. This allows to explore a larger design space.

5.1 Encoding conflicts in a prefix

A CSC conflict can be represented as an unordered *conflict pair* of configurations $\langle C_1, C_2 \rangle$ whose final states are in CSC conflict, as shown if Fig. 2(c). In [10, 11] two techniques for detecting CSC conflicts (based, respectively, on integer programming and SAT) were proposed. Essentially, they allow for efficiently finding such conflict pairs in STG unfolding prefixes.

The set of all conflict pairs may be quite large, e.g., due

to the following 'propagation' effect: if C_1 and C_2 can be expanded by the same event e then $\langle C_1 \cup \{e\}, C_2 \cup \{e\}\rangle$ is also a conflict pair (unless these two configurations enable the same set of local signals). Therefore, it is desirable to reduce the number of pairs needed to be considered, e.g., as follows. A conflict pair $\langle C_1, C_2\rangle$ is called *concurrent* if $C_1 \not\subseteq C_2$, $C_2 \not\subseteq C_1$ and $C_1 \cup C_1$ is a configuration. Below is a slightly modified version of a proposition proven in [10]:

Proposition 4. *Let $\langle C_1, C_2\rangle$ be a concurrent CSC conflict pair. Then $C = C_1 \cap C_1$ is such that either $\langle C, C_1\rangle$ or $\langle C, C_2\rangle$ is a CSC conflict pair.*

Thus concurrent conflict pairs are 'redundant' and should not be considered. The remaining conflict pairs can be classified as follows:

Conflict pairs of type I are such that either $C_1 \subset C_2$ or $C_2 \subset C_1$ (Fig. 2(c) illustrates this type of CSC conflicts).

Conflict pairs of type II are such that $C_1 \setminus C_2 \neq \emptyset \neq C_2 \setminus C_1$ and there exist $e' \in C_1 \setminus C_2$ and $e'' \in C_2 \setminus C_1$ such that $e' \# e''$ (Fig. 7(c) illustrates this type of CSC conflicts).

Definition 6 (Core). Let $\langle C_1, C_2\rangle$ be a conflict pair of configurations. The corresponding *complementary set* is defined as $\mathcal{CS} \stackrel{\text{df}}{=} C_1 \triangle C_2$, where $\triangle$ denotes the symmetric set difference. $\mathcal{CS}$ is a *core* if it cannot be represented as the union of several disjoint complementary sets. $\diamondsuit$

For example, the core corresponding to the conflict pair shown in Fig. 2(c) is $\{e_4, \ldots, e_8, e_{10}\}$ (note that for a conflict pair $\langle C_1, C_2\rangle$ of type I, such that $C_1 \subset C_2$, the corresponding core is simply $C_2 \setminus C_1$), and the core corresponding to the conflict pair $\langle \{e_1, e_4, e_6\}, \{e_2\}\rangle$ in Fig. 7(c) is $\{e_1, e_2, e_4, e_6\}$.

One can show that every complementary set $\mathcal{CS}$ can be partitioned into $C_1 \setminus C_2$ and $C_2 \setminus C_1$, where $\langle C_1, C_2\rangle$ is a conflict pair corresponding to $\mathcal{CS}$. Moreover, if $\mathcal{CS}$ is of type I then one of these parts is empty, while the other is $\mathcal{CS}$ itself. An important property of complementary sets is that for each signal $z \in Z$, the differences between the numbers of z^+– and z^-–labelled events are the same in these two parts (and are 0 if $\mathcal{CS}$ is of type I). This suggests that a complementary set can be eliminated, e.g., by introduction of a new internal signal and insertion of its transition into this set, or by 'dragging' an existing event into it using additional ordering constraints, as these would violate the stated property.

5.2 Core elimination by signal insertion

A framework for visualisation and manual resolution of encoding conflicts was presented in [15], where cores are eliminated by signal insertion. By introducing an additional internal signal and insertion of its transition, say csc^+, into the core one can destroy it eliminating thus the corresponding encoding conflicts. To preserve the consistency of the STG, the transition's counterpart csc^- must also be inserted *outside the core*, in such a way that it is neither concurrent to nor in structural conflict with csc^+. Another restriction is that an inserted signal transitions cannot trigger an input signal transition (the reason is that this would impose constraints on the environment which were not present in the original STG, making it 'wait' for the newly inserted signal).

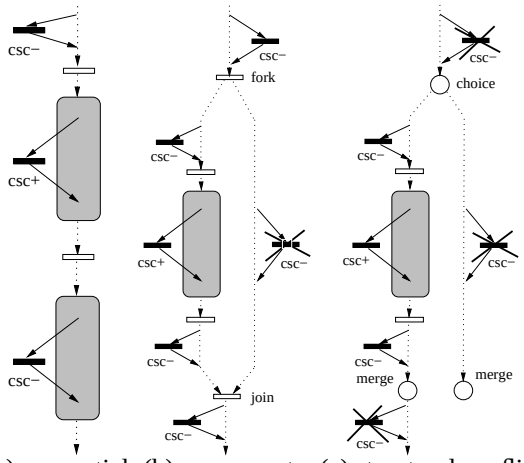
The core in Fig. 2(c) can be eliminated by inserting a new signal, csc^+, somewhere in the core, e.g., concurrently to e_5 and e_6 between e_4 and e_7, and by inserting its complement outside the core, e.g., concurrently to e_{11} between e_9 and e_{12}. (Note that concurrent insertion of these two transitions avoids an increase in the latency of the circuit, where each transition is assumed to contribute a unit delay.) After transferring this signal into the STG, it satisfies the CSC property.

It is often the case that cores overlap. In order to minimise the number of inserted signals, and thus the area and latency of the circuit, it is advantageous to insert a signal in such a way that as many cores as possible are eliminated by it. That is, a signal should be inserted into *the intersection of several cores* whenever possible. In [15] the exploitation of core overlaps is implemented by means of a *height map* showing the quantitative distribution of the cores. The events located in cores are assigned an *altitude*, i.e., the number of cores it belongs to. (The analogy with a topographical map showing the altitudes may be helpful here.) 'Peaks' with the highest altitude are good candidates for insertion, since they correspond to the intersection of maximum number of cores.

The elimination of encoding conflicts by signal insertion is schematically illustrated in Fig. 5, which represent typical cases in STG specifications. Cores 'in sequence', can be eliminated in a 'one-hot' manner as depicted in Fig. 5(a). Each core is eliminated by one signal transition, and its complement is inserted outside the core, preferably, into another non-adjacent one.[1]

An STG that has a core in one of the concurrent branches can also be tackled in a 'one-hot' way, as shown in Fig. 5(b). Note that in order to preserve the consistency, the transition's counterpart cannot be inserted into the concurrent branch, but can be inserted before the fork transition or after the join one. In a branch which is in a structural conflict with another branch, the transition's counterparts must be inserted in the same branch somewhere between the choice

[1] The union of two adjacent cores is usually a complementary set which will not be destroyed if both the transition and its counterpart are inserted into it.



(a) sequential (b) concurrent (c) structural conflict

Figure 5. Strategies for core elimination by signal insertion.

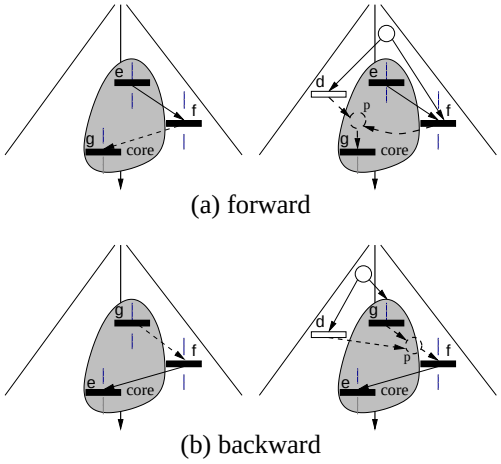


Figure 6. Core elimination by concurrency reduction.

and the merge points, as shown in Fig. 5(c).

Obviously, the described cases do not cover all possible situations and all possible insertions (e.g., one can sometimes insert a new signal transition before the choice point and its counterparts into *each* branch, etc.), but they give an idea how the cores can be eliminated.

5.3 Core elimination by concurrency reduction

Concurrency reduction removes some of the reachable states of the STG and thus can be used for the resolution of encoding conflicts. The elimination of conflict cores by concurrency reduction involves the introduction of additional ordering constraints, which fix some order of execution. In an STG, a fork transition defines the starting point of concurrency and a join transition defines the end point. Existing signals can be used to disambiguate the conflicting states in a core by delaying the starting point or bringing forward the ending point of concurrency. If there is an event concurrent to the core, and a starting or ending point of concurrency is in the core, then this event can be forced into the core by an additional ordering constraint, thus destroying it. For example, in Fig. 2(c), e_9 is concurrent to some of the events in the core, and the starting point of concurrency is in the core, so the concurrency reduction shown by the dashed line in this figure can be used to eliminate the core by 'dragging' e_9 into it. Two kinds of concurrency reduction based transformations for core eliminations are described below.

Forward concurrency reduction illustrated in Fig. 6(a) performs the concurrency reduction $h(E_U) \overset{n}{\dashrightarrow} h(g)$ in the STG, where E_U is a maximal (w.r.t. $\subset$) set of events outside the core which are in structural conflict with each other and concurrent to g — an event in the core. It is assumed that e is in the core, either $e \prec g$ or $e \; co \; g$, and for exactly one event $f \in E_U$, $e \prec f$.

Backward concurrency reduction illustrated in Fig. 6(b) works in a similar way, but the concurrency reduction $h(E_U) \overset{n}{\dashrightarrow} h(f)$ is performed. It is assumed that e is in the core, f is an event outside the core such that $f \prec e$, E_U is a maximal (w.r.t. $\subset$) set of events which are in structural conflict with each other and concurrent to f, such that exactly one event $g \in E_U$ is in the core, and either $g \prec e$ or $g \; co \; e$.

In both cases the core is destroyed by additional ordering constraints 'dragging' f into the core.

These two rules are illustrated by the examples in Fig. 7, where they are applied to cores of types I (parts (a,b) of this figure) and II (parts (c,d) of this figure). In Fig. 7(a) instances of b^+ and a^- are concurrent to the core. The forward concurrency reduction $b^+ \dashrightarrow e^-$ can be applied, because b^+ succeeds e^+ and e^- succeeds e^+. This 'drags' b^+ into the core, destroying it. Note that f is an input and thus cannot be delayed, and so the concurrency reductions $b^+ \dashrightarrow f^+$ and $b^+ \dashrightarrow f^-$ would be invalid. The backward concurrency reductions $e^+ \dashrightarrow a^-$ and $f^+ \dashrightarrow a^-$ can also be applied to eliminate the conflict core, because a^- precedes e^-, and both e^+ and f^+ are in the core and precede e^-. Either of these reductions 'drags' a^- into the core, destroying it.

In Fig. 7(b), d^+ is concurrent to events in the core and precedes c^+, an event in the core. The only event in the core which precedes or is concurrent to c^+ is a^+. However, $a^+ \dashrightarrow d^+$ is an invalid transformation, which introduces a deadlock. The concurrency reduction $\{a^+, b^+\} \dashrightarrow d^+$ has been used instead, since $b^+ \# a^+$ and $b^+ \; co \; d^+$.

Fig. 7(c,d) show the elimination of type II cores. A forward concurrency reduction is illustrated in Fig. 7(c). An instance of d^+ is concurrent to the core and succeeds a^+, an event in the core, and therefore it can be used for a forward reduction. The only possible concurrency reduction is $d^+ \dashrightarrow a^-$, since b^+ is an input and thus cannot be delayed.

The backward concurrency reduction technique is illustrated in Fig. 7(d), where d^+ is concurrent to a^+ and e^+ in the core and precedes b^+ in the core. The only events in the core which either precede or are concurrent to b^+ are a^+ and e^+, and either of them can be used to eliminate the core. However, both reductions $a^+ \dashrightarrow d^+$ and $e^+ \dashrightarrow d^+$ are invalid, since they introduce deadlocks. Thus c^+ should be involved, yielding the following two backward concurrency reductions eliminating the core: $\{a^+, c^+\} \dashrightarrow d^+$ and $\{c^+, e^+\} \dashrightarrow d^+$. Note that the reductions $\{a^+, b^+/1\} \dashrightarrow d^+$ and $\{b^+/1, e^+\} \dashrightarrow d^+$ do not eliminate the core, because d^+ is 'dragged' into both branches of the core, and so the net sum of signals in these two branches remains equal. (And our backward concurrency reduction rule does not allow to use these two transformations, since only one event from the set E_U is allowed to be in the core.)

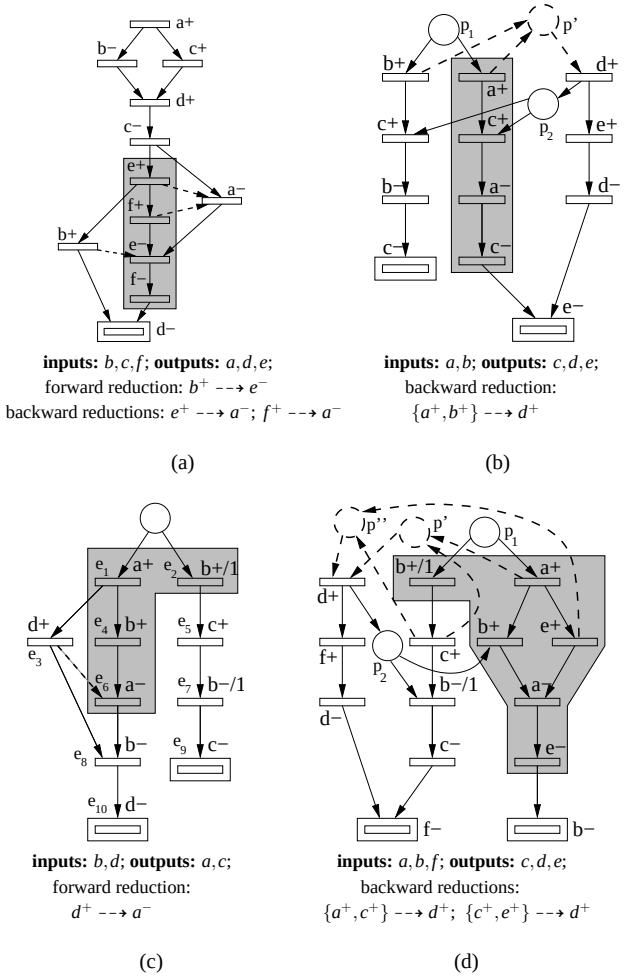


Figure 7. Elimination of type I (a,b) and type II (c,d) cores.

6 Implementation

In our framework, encoding conflicts can be eliminated by the introduction of auxiliary signals and concurrency reduction. A heuristic *cost function* is applied to select the best transformation for the resolution of encoding conflicts. It takes into account: (i) the delay caused by the applied transformation; (ii) the estimated increase in the complexity of the logic and (iii) the number of cores eliminated by the transformation. The resolution process involves finding an appropriate transformation for the elimination of cores in the STG unfolding prefix, as was explained earlier. The following steps are used to resolve the CSC conflicts:

1. Construct an STG unfolding prefix.
2. Compute the cores and, if there are none, terminate.
3. Choose areas for transformation (the 'highest peaks' in the height map corresponding to the overlap of the maximum number of cores are good candidates).
4. Compute valid transformations for the chosen areas and sort them according to the cost function; if no valid transformation is possible then
   - change the transformation areas by including the next highest peak and repeat step 4;
   - otherwise manual intervention by the designer is necessary; the progress might still be possible if the designer relaxes some I/O constraints, uses timing assumptions, etc.
5. Select the best according to the cost function transformation; if it is a signal insertion then the location for insertion of the counterpart transition is also chosen.
6. Perform the best transformation and continue with step 1.

The described framework is being integrated into our tool CONFRES [15].

6.1 Cost function

A cost function was developed to heuristically select on each iteration of the encoding conflict resolution process the best transformation (either a concurrency reduction or a signal insertion). It is comprised of three parts, taking into account the delay penalty inflicted by the transformation, the estimated increase in the complexity of the logic, and the

number of cores eliminated by the transformation:

$$cost \stackrel{\text{df}}{=} \alpha_1 \cdot \Delta\omega + \alpha_2 \cdot \Delta logic + \alpha_3 \cdot \Delta cores \; .$$

The parameters $\alpha_{1,2,3} \geq 0$ are given by the designer and can be used to direct the heuristic search towards reducing the delay inflicted by the transformation (α_1 is large compared with α_2 and α_3) or the estimated complexity of logic (α_2 and α_3 are large compared with α_1).

The first part of the cost function estimates the delay caused by a transformation. A delay model where each transition of the STG is assigned an individual delay is considered; e.g., input signals usually take longer to complete than non-input ones, because they often denote the end of a certain computation in the environment. (This delay model is similar to that in [3,4].) It is quite crude, but it is hard to significantly improve it, since the exact time behaviour is only known after the circuit and its environment are synthesised.

Weighted events' depths in the unfolding prefix are used to determine the delay penalty of the transformation. The weighted depth ω_e of an event e is defined as follows:

$$\omega_e \stackrel{\text{df}}{=} \begin{cases} \omega_{h(e)} & \text{if } {}^\bullet({}^\bullet e) = \emptyset \\ \omega_{h(e)} + \max_{e' \in {}^\bullet({}^\bullet e)} \omega_{e'} & \text{otherwise,} \end{cases}$$

where ω_t is the delay associated with transition $t \in T$. In our implementation, these delays are chosen as follows:

$$\omega_t \stackrel{\text{df}}{=} \begin{cases} 3 & \text{if } t \text{ is an input transition} \\ 1 & \text{otherwise.} \end{cases}$$

These parameters can be fine-tuned by the designer if necessary.

For a concurrency reduction $h(U) \stackrel{n}{\dashrightarrow} t$, the delay penalty $\Delta\omega$ is computed as the difference of the weighted depths of a t-labelled event after and before the concurrency reduction. The value of $\Delta\omega$ is positive if t is delayed by the transformation, otherwise it is 0. Note that the event's depth after the reduction is calculated *using the original prefix*.

For a signal insertion, several (at least two) transitions of a new signal *csc* are added to the STG. For each such a transition t, the inflicted delay penalty $\Delta\omega_t$ is computed, and then these penalties are added up to obtain the total delay penalty $\Delta\omega \stackrel{\text{df}}{=} \sum_t \Delta\omega_t$. If the insertion is concurrent, no additional delay is inflicted ($\Delta\omega_t \stackrel{\text{df}}{=} 0$), since in our delay model the transitions corresponding to internal signals are fast, and so their firing time cannot exceed that of the concurrent transitions. If the insertion is sequential, the inflicted delay penalty $\Delta\omega_t$ is computed by adding up the delay penalties of all the transitions u delayed by t: $\Delta\omega_t \stackrel{\text{df}}{=} \sum_u \Delta\omega_t^u$, where for each such u, the delay penalty $\Delta\omega_t^u$ is computed as the difference of the weighted depths of a u-labelled event after and before the transformation. Note that $\Delta\omega$ is calculated *using the original prefix.*

The second part of the cost function, $\Delta logic$, estimates the increase in the complexity of the logic. The logic complexity is estimated using the number of *triggers* of each local signal. The set of triggers of a signal $z \in Z$ is defined on the (full) unfolding as

$$trg(z) \stackrel{\text{df}}{=} \left\{ z' \in Z \;\middle|\; \exists e' \in \bigcup_{(\lambda \circ h)(e) = z^\pm} trg(e) : (\lambda \circ h)(e') = z'^\pm \right\} ,$$

and can be approximated from below using a finite prefix of the STG unfolding.

We will also denote by $trg'(z)$ the number of triggers of a $z \in Z$ after the transformation. (Note that for the transformations which we use, $trg'(z)$ can be approximated *using the original prefix.*)

For a concurrency reduction $U \stackrel{n}{\dashrightarrow} t$, where t is a z-labelled transition, the estimated increase in complexity of the logic $\Delta logic$ is computed as

$$\Delta logic \stackrel{\text{df}}{=} \mathfrak{C}(|trg'(z)|) - \mathfrak{C}(|trg(z)|) \; ,$$

where

$$\mathfrak{C}(n) \stackrel{\text{df}}{=} \begin{cases} 0 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ \left\lceil \frac{2^n}{n} \right\rceil & \text{if } n > 2 \end{cases}$$

estimates the number of binary gates needed to implement an n-input Boolean function. This formula was chosen because the asymptotic average number of binary gates in a Boolean circuit implementing an n-input Boolean function is $\frac{2^n}{n}$ [18], and that all the triggers of a signal z are always in the support of the complex gate implementing z. Note that the maximal number of triggers which can be added is $|U|$; the actual number of added triggers can be smaller if some of the signals labelling the transitions in U were already triggers of z. (In fact, this number can even be negative, e.g., as in the first solution for the A/D convertor case study described in the next section.) This definition of $\Delta logic$ discourages solutions using complex gates with too many inputs: the penalty is relatively small if the number of triggers is small, but grows exponentially if the transformation adds new triggers to a signal which already had many triggers.

For a signal insertion, several local signals in the modified STG can be triggered by the new signal *csc*. Let Z' denote the set of all such signals. For each signal $z \in Z'$, the increase $\Delta logic_z$ in the complexity of the logic implementing z is estimated, and then these estimates are added up. (Note that $\Delta logic_z$ can be negative for some $z \in Z'$, e.g., when *csc* replaces more than one trigger of z.) Moreover, the added signal *csc* has to be implemented, and thus introduces additional logic complexity, which is estimated and added to the result: $\Delta logic \stackrel{\text{df}}{=} (\sum_{z \in Z'} \Delta logic_z) + \Delta logic_{csc}$, where

$$\Delta logic_z \stackrel{\text{df}}{=} \mathfrak{C}(|trg'(z)|) - \mathfrak{C}(|trg(z)|)$$

for all $z \in Z'$, and

$$\Delta logic_{csc} \stackrel{\text{df}}{=} \mathfrak{C}(|trg'(csc)|) \ .$$

Note that in the case of signal insertion, at most one additional trigger (viz. *csc*) per signal can be introduced.

The third part of the cost function, $\Delta cores$, estimates how many cores are eliminated by the transformation. It is computed by checking for each core 'touched' by the transformation whether it is eliminated or not, *using the original prefix*. While doing this, the following consideration concerning signal insertion should be taken into account: if both rising and falling transitions of the new signal are inserted into the same complementary set, it is not eliminated; in particular, if these transitions are inserted into adjacent cores, the complementary set obtained by uniting these cores will resurface as a new core on the next iteration (even though the original cores are eliminated).

Note that for efficiency reasons the cost function should be computed on the original unfolding prefix. This strategy significantly reduces the number of times the unfolding prefix has to be built, saving time.

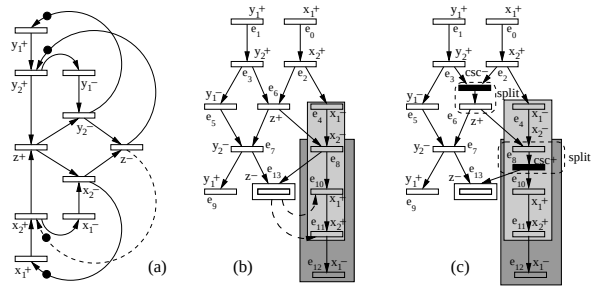
7 Case studies

In this section, two examples demonstrating the proposed combined framework for the resolution of encoding conflicts are discussed.

7.1 Weakly synchronised pipelines

Fig. 8(a) shows an STG modelling two weakly synchronised pipelines without arbitration [11]. The STG exhibits encoding conflicts resulting in two cores shown in Fig. 8(b), where two possible concurrency reductions resolving the CSC conflicts are shown. Both cores can be eliminated by introducing a causal constraint, either $z^- \overset{1}{\dashrightarrow} x_1^+$ or $z^- \overset{1}{\dashrightarrow} x_2^+$. However, the first reduction delays x_1^+ and adds z to the triggers of x_1, whereas the second reduction has no effect on the delay (z^- can be executed concurrently with its predecessor) and on the number of triggers of x_2 (as z^+ already triggers x_2^-). Thus the latter reduction is preferable according to our cost function, resulting in the STG shown in Fig. 8(a), with the dashed arc taken into account. The corresponding equations are presented in Fig. 8(d).

The cores can also be eliminated by an auxiliary signal *csc*. Phase one of the resolution process inserts a signal transition somewhere into the highest peak in the height map, which comprises the events e_8, e_{10} and e_{11}. For example, in Fig. 8(c) a signal transition csc^+ is inserted after e_8 and its counterpart is inserted outside the cores before e_6, ensuring that the cores are destroyed. Other valid insertions are possible, e.g., inserting csc^+ before e_{10} and its counterpart before e_6. Both these transformations eliminate all the cores,



$x_1 = \overline{x_2}$
$x_2 = \overline{z} \cdot (x_1 + x_2) + x_1 \cdot x_2$
$y_1 = \overline{y_2}$
$y_2 = y_1 \cdot (y_2 + \overline{z}) + y_2 \cdot \overline{z}$
$z = y_2 \cdot (z + x_2) + x_2 \cdot z$

(d)

$x_1 = \overline{x_2} \cdot csc$
$x_2 = x_2 \cdot (\overline{z} + csc) + x_1$
$y_1 = \overline{y_2}$
$y_2 = y_1 \cdot (y_2 + \overline{z}) + y_2 \cdot \overline{z}$
$z = z \cdot y_2 + \overline{csc}$
$csc = csc \cdot (\overline{y_2} + z) + \overline{x_2}$

(e)

$x_1 = \overline{x_2} \cdot csc$
$x_2 = x_2 \cdot (\overline{z} + csc) + x_1$
$y_1 = \overline{y_2}$
$y_2 = y_1 \cdot (y_2 + \overline{z}) + y_2 \cdot \overline{z}$
$z = z \cdot y_2 + x_2 \cdot \overline{csc}$
$csc = csc \cdot (\overline{y_2} + z) + \overline{x_2}$

(f)

outputs: x_1, x_2, y_1, y_2, z; **internal:** *csc*

Figure 8. Weakly synchronised pipelines: the STG (a), its unfolding prefix with cores, showing transformations resolving the encoding conflicts (b,c) and the corresponding equations (d,e,f).

and in both of them the newly inserted signal has two triggers, but the former insertion delays three transitions, adds the trigger *csc* to x_1 and replaces the trigger x_2 of z with *csc*, whereas the latter insertion delays two transitions and adds the trigger *csc* to x_1 and z. The equations corresponding to these two solutions are shown in Fig. 8(e,f).

One can see that the implementations derived by signal insertion are more complex than the one obtained by concurrency reduction. These two implementations also delay signals z and x_1, whereas the one derived using concurrency reduction does not have additional delays. Additionally, the solution obtained by concurrency reduction results in a symmetrical STG.

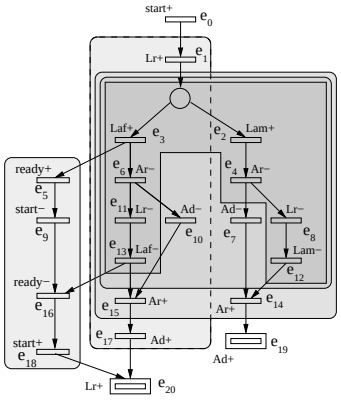
7.2 A/D converter

The example shown in Fig. 9 is a part of the A/D converter proposed in [13]. It contains two type I and three type II cores shown in Fig. 9(a). The events e_3, e_6, e_{11} and e_{13} comprise the highest peak, as each of them belongs to four cores. They can be eliminated by a forward concurrency reduction, since events e_5 and e_6 are concurrent to the events in the peak and the concurrency starts in the peak. The valid concurrency reductions are presented in the table in Fig. 9(c), where the column 'lits' shows the total number of literals in the corresponding equations. The first four solutions eliminate all the cores in the peak, and the last one eliminates only one core. Incidentally, the first four solutions eliminate the remaining core as well, because the corresponding ordering constraints also act as backward

concurrency reductions. The first solution introduces a large delay (e_{11} is delayed by an input event e_9) but no additional triggers (in fact, the number of triggers of *Lr* is reduced, since *Ar* ceases to be its trigger), whereas the second one does not delay e_{11} but introduces an additional trigger. The equations for these two solutions are shown in Fig. 9(b). The third solution delays e_6 by e_5, and the fourth solution delays e_6 by e_5 and e_9; moreover, both these solutions introduce an additional trigger to *Ar* (which already had three triggers), and thus are inferior according to our cost function.

Alternatively, the encoding conflicts can be solved using signal insertion, by inserting a transition csc^+ into the peak and its counterpart outside the cores belonging to the peak, preserving the consistency and ensuring that the cores are destroyed. Recall that input signal transitions cannot be delayed by newly inserted transitions, i.e., in the peak csc^+ cannot delay e_3 and e_{13}. In the second phase, the parts of the prefix which are concurrent to or in structural conflict with the inserted transition are faded out, as the consistency would be violated if the csc^- is inserted there. At the same time, one can try to eliminate the remaining core $\{e_5, e_9, e_{16}, e_{18}\}$. The valid signal insertions are shown in the table in Fig. 9(d), where a sequential insertion is designated by either 'after' or 'before' e_i, inserting a signal transition directly either after or before the transition corresponding to e_i. A concurrent signal transition insertion is denoted by $\| e_i$ to e_j, where a signal transition is inserted between e_i and e_j.

Solution 6 introduces the smallest delay (only $ready^-$ is delayed), whereas solution 7 has the smallest estimated logic complexity, but the largest delay (the insertion delays $ready^+$, Ar^- and $ready^-$). Solutions 9 and 11 have the greatest estimated logic complexity. The equations for solution 6, 7 and 9 are presented in Fig. 9(b). One can see that the equations for solution 7 and 9 have the same number of literals, even though their estimated logic complexities were quite different. This shows that our cost function is not perfect, since $\mathfrak{C}(n)$ is quite a rough estimate of complexity, and since the cost function does not take the context signals into account. However, it is not trivial to significantly improve this cost function without introducing a considerable time overhead in computing it. (In particular, the context signals cannot be computed for a particular signal *z* until all the encoding conflicts for *z* are resolved.)



(a)

$$
\begin{aligned}
&ready = Laf\\
&\quad Lr = start \cdot Ad \cdot Ar +\\
&\qquad Laf \cdot (Ar + \overline{ready})\\
&\quad Ar = \overline{Lam} \cdot \overline{Laf} \cdot (Ar + \overline{Ad})\\
&\quad \text{equations for solution 1}\\
&ready = start \cdot ready + Laf\\
&\quad Lr = start \cdot Ad \cdot \overline{ready} \cdot Ar +\\
&\qquad Laf \cdot (Ar + \overline{ready})\\
&\quad Ar = \overline{Lam} \cdot \overline{Laf} \cdot (Ar + \overline{Ad})\\
&\quad \text{equations for solution 2}\\
&ready = Laf + csc\\
&\quad Lr = start \cdot Ad \cdot Ar \cdot \overline{csc} +\\
&\qquad Laf(\overline{csc} + Ar)\\
&\quad Ar = \overline{Lam} \cdot \overline{Laf} \cdot (Ar + \overline{Ad})\\
&\quad csc = start \cdot csc + Laf\\
&\quad \text{equations for solution 6}\\
&ready = csc\\
&\quad Lr = Ar \cdot (start \cdot \overline{csc} \cdot Ad + Laf)\\
&\quad Ar = \overline{Lam} \cdot \overline{Laf} \cdot (Ar + \overline{Ad}) +\\
&\qquad Laf \cdot \overline{csc}\\
&\quad csc = start \cdot csc + Laf\\
&\quad \text{equations for solution 7}\\
&ready = Laf + csc\\
&\quad Lr = \overline{csc} \cdot (start \cdot Ar \cdot Ad + Laf)\\
&\quad Ar = \overline{Lam} \cdot \overline{Laf} \cdot (Ar + \overline{Ad})\\
&\quad csc = start \cdot csc + Laf \cdot \overline{Ar}\\
&\quad \text{equations for solution 9}
\end{aligned}
$$

(b)

concurrency reduction			
#	causal constraint	Δω;Δ*logic*;Δ*cores*	lits
1	$h(e_9) \dashrightarrow h(e_{11})$	3;-1;-5	11
2	$h(e_5) \dashrightarrow h(e_{11})$	0;2;-5	14
3	$h(e_5) \dashrightarrow h(e_6)$	1;2;-5	14
4	$h(e_9) \dashrightarrow h(e_6)$	4;2;-5	11
5	$h(e_{10}) \dashrightarrow h(e_{11})$	3;0;-1	n/a

(c)

signal insertion				
#	phase 1	phase 2	Δω;Δ*logic*;Δ*cores*	lits
6	$\| e_3$ to e_{11}	before e_{16}	1;3;-5	16
7	after e_3	before e_{16}	3;2;-5	15
8	before e_6	before e_{16}	2;3;-5	16
9	before e_{11}	before e_{16}	2;4;-5	15
10	after e_3	after e_9	3;3;-5	18
11	after e_3	$\| e_5$ to e_{16}	2;4;-5	20

(d)

inputs: *start*, *Lam*, *Laf*, *Ad*; **outputs:** *ready*, *Lr*, *Ar*; **internal:** *csc*

Figure 9. Top level of the A/D converter: the unfolding prefix with cores (a), a selection of equations corresponding to valid transformations (b), and lists of possible concurrency reductions (c) and signal insertions (d).

8 Conclusions and future work

This paper presents a combined framework for the resolution of encoding conflicts in STG unfoldings. This framework explores a larger design space and allows the designer to exploit the area/delay tradeoff, which is crucial in synthesis of many interface controllers, e.g., in the 'glue logic' between IP cores of SoCs. Encoding conflicts are represented by means of cores, which are sets of transitions 'causing' them. The advantage of using cores is that only those parts of STGs which cause encoding conflicts, rather than the complete list of CSC conflicts, are considered. Since the number of cores is usually much smaller than the number of encoding conflicts, this approach reduces the amount of information to be analysed.

Moreover, a novel validity condition has been proposed to justify these transformations, which is also of independent interest. We have developed a sufficient condition for

a concurrency reduction on a general LPN being valid, as well as a simplified version of this condition for the case of a non-auto-concurrent Petri net.

The future work will be focused on the following issues:

- developing an algorithm for checking the validity of a concurrency reduction and a signal insertion on safe nets;
- improving the cost function;
- performing the transformations directly on the unfolding prefix rather than the STG whenever possible, in order to reduce the number of runs of the unfolding algorithm.

Acknowledgements

We would like to thank Maciej Koutny and Walter Vogler for helpful comments and suggestions. This research was supported by the EPSRC grants GR/R16754 (BESST) and GR/S12036 (STELLA).

References

[1] J. Carmona, J. Cortadella and E. Pastor: A Structural Encoding Technique for the Synthesis of Asynchronous Circuits. *Fundamentae Informaticae* 50(2) (2002) 135–154.

[2] T. -A. Chu: *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD Thesis, MIT/LCS/TR-393 (1987).

[3] 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) (1997) 315–325.

[4] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno and A. Yakovlev: *Logic Synthesis of Asynchronous Controllers and Interfaces*. Springer Verlag (2002).

[5] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno and A. Yakovlev: Automatic Handshake Expansion and Reshuffling Using Concurrency Reduction. Proc. of *HWPN'98*, (1998) 86–110.

[6] D. L. Dill: *Trace Theory for Automatic Heirarchical Verification of Speed-Independent Circuits*. PhD Thesis 15213, CMU (1987).

[7] J. Engelfriet: Branching Processes of Petri Nets. *Acta Informatica* 28 (1991) 575–591.

[8] J. Esparza, S. Römer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. *Formal Methods in System Design* 20(3) (2002) 285–310.

[9] V. Khomenko: *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD Thesis, School of Computing Science, University of Newcastle upon Tyne (2003).

[10] V. Khomenko, M. Koutny and A. Yakovlev: Detecting State Coding Conflicts in STGs Using Integer Programming. Proc. of *DATE'02*, IEEE Computer Society Press (2002) 338–345.

[11] V. Khomenko, M. Koutny and A. Yakovlev: Detecting State Coding Conflicts in STG Unfoldings Using SAT. Proc. of *ACSD'03*, IEEE Computer Society Press (2003) 51–60. Full version: Special Issue on Best Papers from ACSD'03, Fundamentae Informaticae 62(2) (2004) 1-21.

[12] V. Khomenko, M. Koutny and A. Yakovlev: Logic Synthesis for Asynchronous Circuits Based on Petri Net Unfoldings and Incremental SAT. Proc. of *ACSD'04*, IEEE Computer Society Press (2004) 16–25. Full version: to appear in Special Issue on Best Papers from ACSD'04, Fundamentae Informaticae.

[13] D. J. Kinniment, B. Gao, A. Yakovlev and F. Xia: Towards asynchronous A-D conversion. Proc. of *ASYNC'00*, IEEE Computer Society Press (2000) 206–215.

[14] B. Lin, C. Ykman-Couvreur and P. Vanbekbergen: A General State Graph Transformation Framework for Asynchronous Synthesis. Proc. of *EURO-DAC'94*, IEEE Computer Society Press (1994) 448–453.

[15] A. Madalinski, A. Bystrov, V. Khomenko and A. Yakovlev: Visualization and Resolution of Coding Conflicts in Asynchronous Circuit Design. Proc. of *DATE'03*, IEEE Computer Society Press (2003) 926–931. Full version: Special Issue on Best Papers from DATE'2003, IEE Proceedings: Computers & Digital Techniques 150(5) (2003) 285–293.

[16] H. Saito, A. Kondratyev, J. Cortadella, L. Lavagno and A. Yakovlev: What Is the Cost of Delay Insensitivity? Proc. of *CAD*, IEEE Computer Society Press (1999) 316–323.

[17] A. Semenov: *Verification and Synthesis of Asynchronous Control Circuits Using Petri Net Unfolding*. PhD Thesis, University of Newcastle upon Tyne (1997).

[18] I. Wegener: *The Complexity of Boolean Functions*. Wiley-Teubner Series in Computer Science (1987).