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

Victor Khomenko[1] Agnes Madalinski[2] Alex Yakovlev[2]
[1]School of Computing Science,
[2]School of Electrical, Electronic and Computer Engineering,
University of Newcastle upon Tyne, UK.
E-mail: {Victor.Khomenko,A.A.Madalinski,Alex.Yakovlev}@ncl.ac.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. The method has been implemented in the CONFRES tool and applied to a number of case studies. The experimental results show that the combined framework enlarges the design space and allows better exploration of the speed/area tradeoff.**

## I. INTRODUCTION

SIGNAL Transition Graphs, or STGs [2], 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: (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 [2], 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 hampers 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.

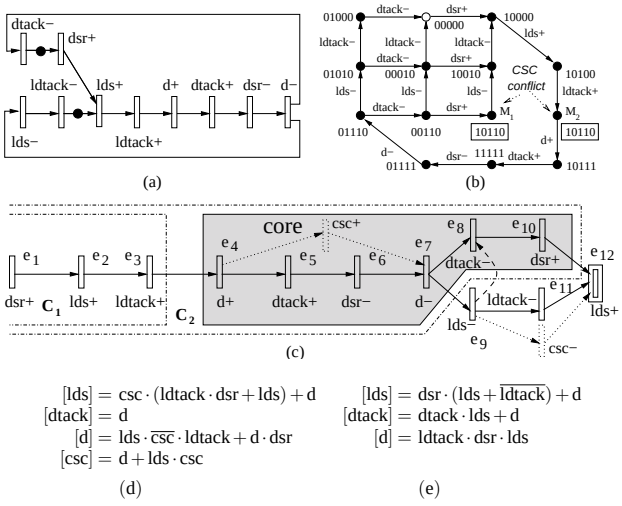
Where PETRIFY fails, other tools based on alternative techniques, and in particular those employing Petri net unfoldings, may succeed. A *finite and complete unfolding prefix* of an STG $\Gamma$ is a finite acyclic net which implicitly represents all the reachable states of $\Gamma$ together with transitions enabled at those states. Intuitively, it can be obtained through *unfolding* $\Gamma$, by successive firings of transition, under the following assumptions: (i) for each new firing a fresh transition (called an *event*) is generated; (ii) for each newly produced token a fresh place (called a *condition*) is generated. The unfolding is infinite whenever $\Gamma$ has an infinite run; however, if $\Gamma$ 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. 1(c) shows a finite and complete unfolding prefix (with the only cut-off event depicted as a double box) of the STG shown in Fig. 1(a).

Efficient algorithms exist for building such prefixes [6], which ensure that the number of non-cut-off events in a complete prefix can never exceed the number of reachable states of $\Gamma$. 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 $2^{100}$ vertices, whereas the complete prefix will coincide with the net itself.

Since practical 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 [6], [7] they were 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 [6], [7] the unfolding technique was applied to the implementability analysis in step (a), viz. checking the Complete State Coding (CSC) condition [2], which requires detecting CSC conflicts between reachable states of an STG. A CSC conflict arises when semantically different reachable states of an STG have the same binary encoding. Fig. 1(b) shows the state graph of the STG in Fig. 1(a) with a CSC conflict between states $M_1$ and $M_2$.

In [12] 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. There a




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

Fig. 1. 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*.

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 [8] addresses step (c), using unfolding techniques to derive equations for logic gates of the circuit. The results in [6]–[8], [12] form a complete design flow for complex-gate synthesis of asynchronous circuits based on STG unfoldings rather than state graphs.

The resolution of encoding conflicts by signal insertion is illustrated in Fig. 1(c), where the new signal *csc* is helping to distinguish between the states involved in the encoding conflict. (Intuitively, insertion of signals introduces additional memory into the circuit, helping to trace the current state.) It was inserted concurrently to existing transitions in order to minimise the latency, and in such a way that the 'external' behaviour of the STG does not change. Alternatively, the encoding conflict can be resolved by reducing the concurrency between $lds^-$ and $dtack^-$ (as shown by the dashed arc in Fig. 1(c)) so that state $M_1$ is removed from the reachability graph shown in Fig. 1(b), which in turn resolves the encoding conflict. The logic equations corresponding to these solutions are shown in Fig. 1(d,e).

One can see that in this example the equations for the signal insertion are more complex then those for concurrency reduction. It is often the case that concurrency reduction produces smaller circuits, which may also be 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.

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 latency 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. On the other hand, there are situations when signal insertion produces better solutions. 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 [3], [11] or are restricted to specific net classes or use local transformations [1] and thus restrict the design space.

This paper extends the framework for the visualisation and resolution of encoding conflicts in [12] (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. This allows one to explore a larger 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 [14] to convert speed-independent circuits into delay-insensitive ones.

This paper aims at presenting these results in a relatively informal way. More formal presentation can be found in technical report [9].

## II. Valid transformations

For the sake of generality, we discuss arbitrary labelled Petri nets (LPNs) (STGs being a special kind of them). That is, there are disjoint sets of inputs $I$ and outputs $O$, and a function $\ell$ mapping the transitions of the Petri net to the set $I \cup O \cup \{\tau\}$, where $\tau \notin I \cup O$ is a *silent action* (e.g., internal signals in an STG), which is not observable by the environment. In figures, we will denote inputs by $i$ or $i_k$, and outputs by $o$ or $o_k$. We assume that the transformation does not change the inputs and outputs of the system, and we will denote by $\Upsilon$ and $\Upsilon'$ the original and transformed LPNs, respectively.

Given an LPN $\Upsilon$, a set of its transitions $U \neq \emptyset$, its transition $t \notin U$ and $n \in \mathbb{N}$, a *concurrency reduction* $U \overset{n}{\dashrightarrow} t$ is defined as the transformation adding to $\Upsilon$ a new place $p$, which initially has $n$ tokens, the arc $(u,p)$ for each transition $u \in U$ and the arc $(p,t)$, as shown in Fig. 2. We will write $U \dashrightarrow t$ instead of $U \overset{0}{\dashrightarrow} t$ and $u \overset{n}{\dashrightarrow} t$ instead of $\{u\} \overset{n}{\dashrightarrow} t$. Note that concurrency reduction cannot add new behaviour to the system — it can only restrict it.

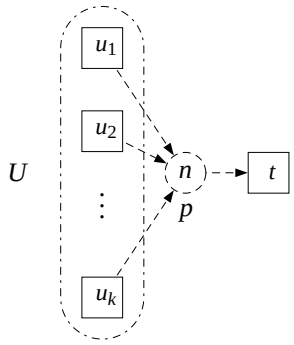



Fig. 2. Concurrency reduction $U \xrightarrow{n} t$.

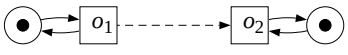
The notion of validity for signal insertion is straightforward — one can justify such a transformation in terms of weak bisimulation, which is well-studied. For a concurrency reduction (or transformations in general), the situation is more difficult: the original and transformed systems are typically 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 [14], 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 [4], [14], 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 boundedness and speed-independence of the STG [2], [3].
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 is valid.



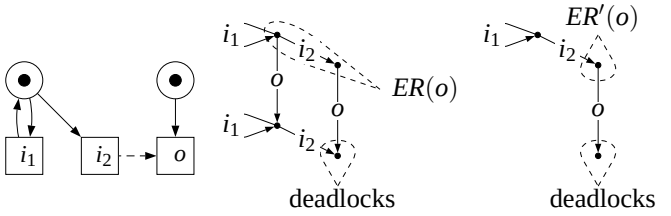




Fig. 3. Liveness problem: an LPN with the concurrency reduction shown by the dashed arc together with its state graphs before and after the transformation.

In this section we discuss in the described framework the notions of validity proposed in [3], [4], [15] 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 [2], we will concentrate on the remaining two aspects, viz. conformation and liveness.

### A. Overview of previous validity notions

The liveness restrictions imposed on transformations in [3] 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 [4] a notion of *conformation* was 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 [4] is based on the existence of a winning strategy in a certain infinite game, and is quite complicated.

In [15] a notion of a *valid implementation* was introduced, which can be used to justify the signal insertion transformation and takes care of liveness. Moreover, it allows the implementation to have additional inputs with arbitrary behaviour following, because the implementation will work in an environment that, according to the specification, will never produce these inputs. This in particular allows more concurrency for inputs, and can be used to justify the concurrency-increasing transformation of [14]. However, this feature is not complemented by allowing to decrease the concurrency for outputs, and thus this notion cannot be used to justify the concurrency reduction transformation.

In [5] a notion of refinement was introduced, which works well for *deterministic* processes, i.e., CSP processes without the non-deterministic choice operator $\sqcap$. Moreover, a hybrid operator combining parallel composition and hiding was proposed, which preserves determinacy. However, for non-deterministic circuits (e.g., those containing arbiters) a potentially interesting behaviour can be lost due to the following effect: the process $o_1 \sqcap o_2$ (which can be obtained by hiding some of the signals in a circuit with arbitration) can be refined

to $o_1$, i.e., a branch of arbitration can be refined away. Though the observer interacting with $o_1$ is unable to tell that he is not interacting with $o_1 \sqcap o_2$, this may be not what the designer intended. (In [16] the issue of static vs. dynamic determinism is raised in the context of delay-insensitive CSP processes.)

In this paper we propose a relatively simple bisimulation-style notion which takes the liveness into account and allows one to justify both reduction of concurrency for outputs and increase of concurrency for inputs, as well as signal insertion.

### B. Our notion of validity

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 by linearisation of the corresponding partial order. An LPN 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. We will view processes as subnets of the unfolding (and the unfolding can be considered as an overlay of processes). Hence the nodes of a process are not anonymous entities, but correspond to the nodes of the unfolding. We will define by $\sqsubseteq$ the *prefix relation* on processes. Note that the fact that the nodes of a process have identities matters, e.g., if $\pi \sqsubseteq \pi'$ then $\pi'$ is a continuation of $\pi$ rather than some unrelated to $\pi$ process whose initial part is isomorphic to $\pi$. A process 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.

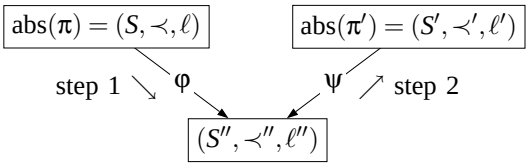
If $\pi$ is a process, we denote by $\mathrm{abs}(\pi)$ the *abstraction* of $\pi$, i.e., the labelled partially ordered set $(S, \prec, \ell)$ (with the labels in $I \cup O$) obtained from the non-$\tau$-labelled events of $\pi$ and the appropriately restricted causal ordering of the events of $\pi$. The elements of $\mathrm{abs}(\pi)$, unlike the nodes of $\pi$, are considered anonymous. (Labelled partially ordered sets are essentially the *pomsets* of [13].)

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 $s \in S$.

Given processes $\pi$ of $\Upsilon$ and $\pi'$ of $\Upsilon'$, we define a relation between their abstractions, $\mathrm{abs}(\pi)$ and $\mathrm{abs}(\pi')$, which holds iff in $\pi'$ the inputs are no less concurrent and the outputs are no more concurrent than in $\pi$. 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 conform (in the sense of [4]) to the original specification.

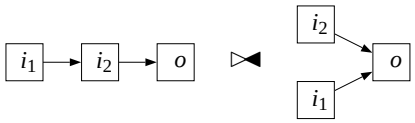
The definition below assumes the *weak fairness*, i.e., that a transition cannot remain enabled forever: it must either fire or be disabled by another transition firing. In particular, this guarantees that the expected inputs eventually arrive, and thus the concurrency reduction $i \dashrightarrow o$ cannot be declared invalid just because the input $i$ fails to arrive and so the output $o$ is never produced.

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 $\prec'$).

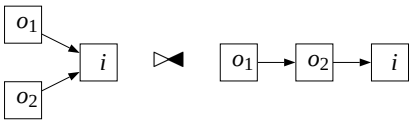



*Definition 1:* Let $\pi$ and $\pi'$ be processes of $\Upsilon$ and $\Upsilon'$, respectively, $\mathrm{abs}(\pi) = (S, \prec, \ell)$ and $\mathrm{abs}(\pi') = (S', \prec', \ell')$. We define $\mathrm{abs}(\pi) \bowtie \mathrm{abs}(\pi')$ if there exist a labelled partially ordered set $(S'', \prec'', \ell'')$ 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);

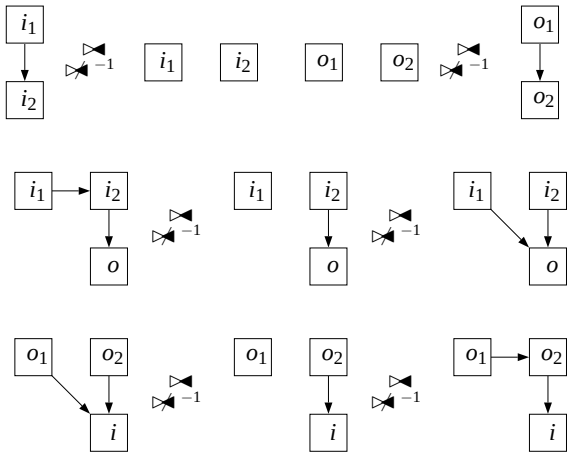


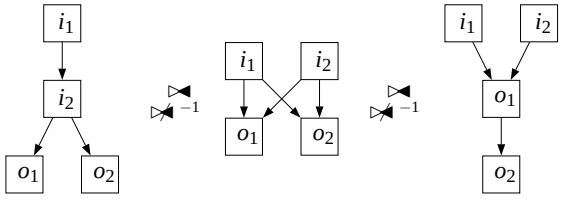

- if $e'$ is an input event and $f' \in DP_{\prec'}(e')$ then $\psi(f') \in DP_{\prec''}\psi(e')$ (in step 2, no new *direct* ordering constraints for inputs can appear, and existing ones can become indirect, e.g., as in the picture below).




◊

*Example 1:* The following hold:






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

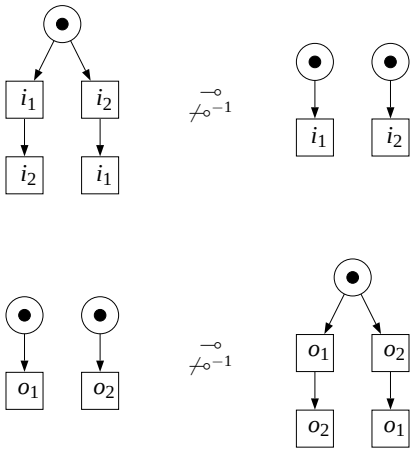
*Definition 2 (Validity):* $\Upsilon'$ is a *valid realisation* of $\Upsilon$, denoted $\Upsilon \multimap \Upsilon'$, if there is a relation $\propto$ between the finite processes of $\Upsilon$ and $\Upsilon'$ such that $\pi_\emptyset \propto \pi'_\emptyset$ (where $\pi_\emptyset$ and $\pi'_\emptyset$ are the empty processes of $\Upsilon$ and $\Upsilon'$, respectively), and for all finite processes $\pi$ and $\pi'$ such that $\pi \propto \pi'$:

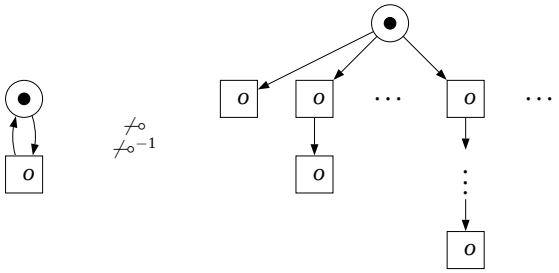
- $\pi \bowtie \pi'$
- For all maximal processes $\Pi' \sqsupseteq \pi'$, and for all finite processes $\widehat{\pi}' \sqsupseteq \pi'$ such that $\widehat{\pi}' \sqsubseteq \Pi'$, there exist finite processes $\widetilde{\pi}' \sqsupseteq \widehat{\pi}'$ and $\widetilde{\pi} \sqsupseteq \pi$ such that $\widetilde{\pi}' \sqsubseteq \Pi'$ 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 $\Upsilon$ is *eventually* performed by $\Upsilon'$ (up to the $\bowtie$ relation) and cannot be pre-empted due to choices, and vice versa, i.e., $\Upsilon'$ and $\Upsilon$ simulate each other with a finite delay. Note that $\multimap$ is a pre-order, i.e., $\Upsilon \multimap \Upsilon$ and 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 invalid, 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 $\bowtie$ relation) process in the transformed LPN, which would have to fire $i_2$ before it is able to fire $o$.

*Example 2:* The following hold:




To show why the relations $\multimap$ and $\multimap^{-1}$ do not hold in the last case we proceed as follows. Suppose one of these relations holds. Then the empty processes of the two systems must be bound by the corresponding $\propto$ relation. Since the process comprised of the leftmost event of the second system cannot be extended (note that the nodes of the process have identities), it must be bound by $\propto$ relation to the process of the first system comprising one instance of $o$. However, the latter process has extensions which cannot be matched (in the sense of $\bowtie$ or $\bowtie^{-1}$) by the former one (since it has no extensions), a contradiction. $\Diamond$

In the full version of this paper [9], we propose criteria which can be applied to check the validity of a concurrency reduction.

## III. Resolution of encoding conflicts

At the level of unfoldings, encoding conflicts can be compactly represented using conflict *cores* [12]. Encoding conflicts can be resolved by either adding auxiliary signals or by concurrency reduction. The former approach was studied in [12], 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.

Due to its structural properties (such as acyclicity), the reachable states of an STG can be represented using *configurations* of its unfolding. A configuration $C$ is a downward-closed set of events (being downward-closed means that if $e \in C$ and $f$ is a causal predecessor of $e$ then $f \in C$) without *structural conflicts* (i.e., for all distinct events $e, f \in C$, there is no condition $c$ in the unfolding such that the arcs $(c,e)$ and $(c,f)$ are in the unfolding). For example, in Fig. 8(a) $\{e_0, e_1, e_2, e_4, e_7, e_8\}$ is a configuration whereas $\{e_0, e_1, e_2, e_3, e_4\}$ and $\{e_0, e_2\}$ are not (the former includes a structural conflict between the events $e_2$ and $e_3$, while the latter does not include $e_1$, a causal predecessor of $e_2$). Intuitively, a configuration is a partial-order execution, i.e., an execution where the order of firing of concurrent events is not important; e.g., the configuration $\{e_0, e_1, e_2, e_4, e_7, e_8\}$ corresponds to two totally ordered executions: $e_0e_1e_2e_4e_7e_8$ and $e_0e_1e_2e_4e_8e_7$. Configurations are somewhat similar to processes, the difference being that the former are sets of events of the unfolding while the latter are nets.

### A. 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. 1(c). In [6], [7] 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 output and internal 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 [6]:

*Proposition 1:* 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. 1(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 $e'$ and $e''$ are in structural conflict (Fig. 6(c) illustrates this type of CSC conflicts).

*Definition 3 (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. A complementary set is of type I/II if the corresponding conflict pair is of type I/II, respectively. $\Diamond$

For example, the core corresponding to the conflict pair shown in Fig. 1(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 complementary set 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. 6(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.

It is often the case that cores overlap. In order to minimise the number of performed transformations, and thus the area and latency of the circuit, it is advantageous to perform such a transformation that as many cores as possible are eliminated by it. That is, a transformation should be performed in the *intersection of several cores* whenever possible. In [12] the exploitation of core overlaps is implemented by means of a *height map* showing the quantitative distribution of the cores (see Fig. 8(b)). Each event in the prefix is 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.

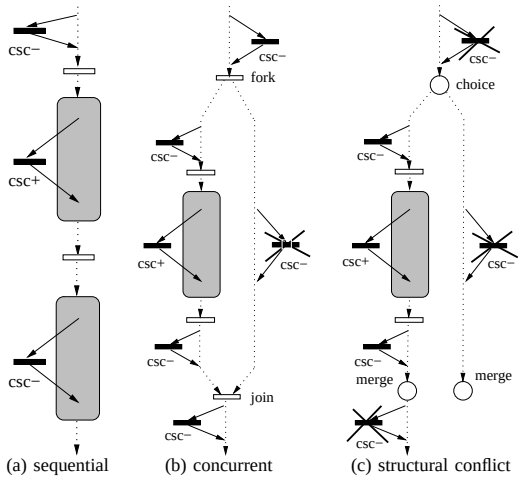



Fig. 4. Strategies for core elimination by signal insertion.

### B. Core elimination by signal insertion

A framework for visualisation and manual resolution of encoding conflicts was presented in [12], where cores were 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 must not 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. 1(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.

The elimination of encoding conflicts by signal insertion is schematically illustrated in Fig. 4, which represent typical cases in STG specifications [12].

### C. 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

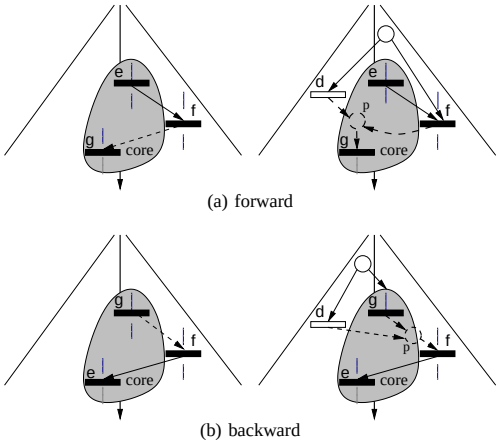



Fig. 5. Core elimination by concurrency reduction.

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 its ending 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. 1(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 elimination are described below (where $h$ is the mapping from the nodes of the prefix to the nodes of the STG).

**Forward concurrency reduction** illustrated in Fig. 5(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 and either precedes $g$ or is concurrent to $g$, and for exactly one event $f \in E_U$, $e$ precedes $f$.

**Backward concurrency reduction** illustrated in Fig. 5(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 $f$ precedes $e$, $E_U$ 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 $g$ either precedes $e$ or is concurrent to $e$.

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. 6, 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. 6(a) instances of $b^+$ and $a^-$ are concurrent to the core. The forward concurrency reduction $b^+ \dashrightarrow e^-$ can be applied, because $b^+$

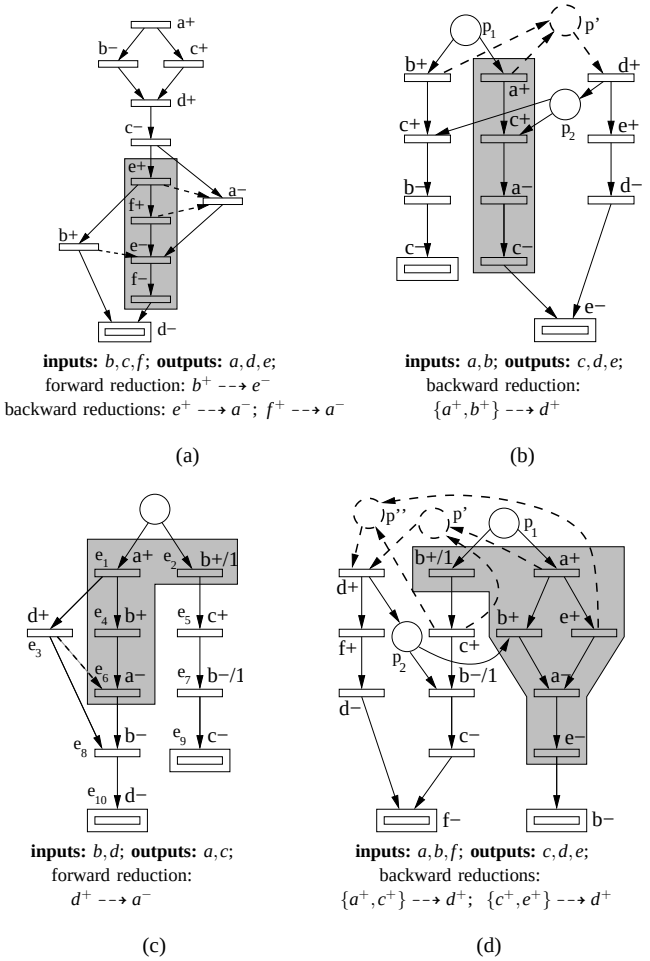



Fig. 6. Elimination of type I (a,b) and type II (c,d) cores.

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. 6(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. (Note that this transformation is ruled out by the maximality requirement in the definition of a backward concurrency reduction.) The forward concurrency reduction $\{a^+, b^+\} \dashrightarrow d^+$ has been used instead, since $b^+$ is in structural conflict with $a^+$ and concurrent to $d^+$.

Fig. 6(c,d) show the elimination of type II cores. A forward concurrency reduction is illustrated in Fig. 6(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. 6(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. (Again, these transformations are ruled out by the maximality requirement in the definition of a backward concurrency reduction.) 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.)

### D. Implementation

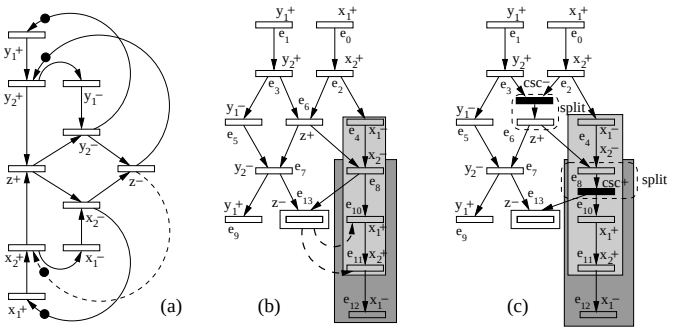
In our framework, encoding conflicts can be eliminated by the introduction of auxiliary signals and concurrency reduction. A heuristic *cost function* is applied to select the best transformation for the resolution of encoding conflicts. It has the form

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

and takes into account: (i) the estimated delay $\Delta\omega$ caused by the applied transformation; (ii) the estimated increase in the complexity of the logic $\Delta logic$ (computed using the *triggers* of each output and internal signal; note that a signal's triggers are guaranteed to be in the support of this signal); and (iii) the number of cores eliminated by the transformation, $\Delta cores$. 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 ($\alpha_1$ is large compared with $\alpha_2$ and $\alpha_3$) or the estimated complexity of logic ($\alpha_2$ and $\alpha_3$ are large compared with $\alpha_1$). This cost function is computed using the unfolding prefix, without synthesising the circuit (see [9] for more details).

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.




$$\begin{aligned} [x_1] &= \overline{x_2} \\ [x_2] &= \overline{z} \cdot (x_1 + x_2) + x_1 \cdot x_2 \\ [y_1] &= \overline{y_2} \\ [y_2] &= y_1 \cdot (y_2 + \overline{z}) + y_2 \cdot \overline{z} \\ [z] &= y_2 \cdot (z + x_2) + x_2 \cdot z \end{aligned}$$

(d)

$$\begin{aligned} [x_1] &= \overline{x_2} \cdot \mathrm{csc} \\ [x_2] &= x_2 \cdot (\overline{z} + \mathrm{csc}) + x_1 \\ [y_1] &= \overline{y_2} \\ [y_2] &= y_1 \cdot (y_2 + \overline{z}) + y_2 \cdot \overline{z} \\ [z] &= z \cdot y_2 + \overline{\mathrm{csc}} \\ [\mathrm{csc}] &= \mathrm{csc} \cdot (\overline{y_2} + z) + \overline{x_2} \end{aligned}$$

(e)

$$\begin{aligned} [x_1] &= \overline{x_2} \cdot \mathrm{csc} \\ [x_2] &= x_2 \cdot (\overline{z} + \mathrm{csc}) + x_1 \\ [y_1] &= \overline{y_2} \\ [y_2] &= y_1 \cdot (y_2 + \overline{z}) + y_2 \cdot \overline{z} \\ [z] &= z \cdot y_2 + x_2 \cdot \overline{\mathrm{csc}} \\ [\mathrm{csc}] &= \mathrm{csc} \cdot (\overline{y_2} + z) + \overline{x_2} \end{aligned}$$

(f)

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

Fig. 7. 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).

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 selected transformation and continue with step 1.

The described framework has been integrated into our tool CONFRES [12].

## IV. CASE STUDIES

In this section, two examples demonstrating the proposed combined framework for the resolution of encoding conflicts are discussed. The simulation times are obtained by the analog simulation using the AMS-0.35$\mu$ CMOS technology.

### A. Weakly synchronised pipelines

Fig. 7(a) shows an STG modelling two weakly synchronised pipelines without arbitration [7]. Note that in this STG all the signals are considered outputs, i.e., the control logic is designed as a closed circuit. (The inputs are inserted after the synthesis is completed, by breaking up some outputs and inserting the environment into the breaks, thus forming a handshake). Hence, in our simulation we measured the *minimum cycle time*, i.e., the time needed to fire all the transitions once in the maximally concurrent way, rather than input-to-output delays.

The STG exhibits encoding conflicts resulting in two cores shown in Fig. 7(b), where two possible concurrency reductions resolving the CSC conflicts are shown. Both cores can be eliminated by introducing a causal constraint, either $z^- \stackrel{1}{\dashrightarrow} x_1^+$ or $z^- \stackrel{1}{\dashrightarrow} x_2^+$. 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, and the simulation results are in good agrement with it: the minimum cycle time for the former concurrency reduction is 2.5ns, and for the latter one (shown in Fig. 7(a) by the dashed line) it is 2.3ns. The corresponding equations for the latter solution are presented in Fig. 7(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. 7(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, 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 $z$. Hence, according to our cost function, the former solution is slower but has a slightly simple logic, which is in good agreement with the resulting implementation and the corresponding simulation results: the equations for the former implementation (see Fig. 7(e)) have 19 literals and its the minimum cycle time is 3.5ns, and the equations for the latter implementation (see Fig. 7(f)) have 20 literals and its minimum cycle time is 3.2ns.

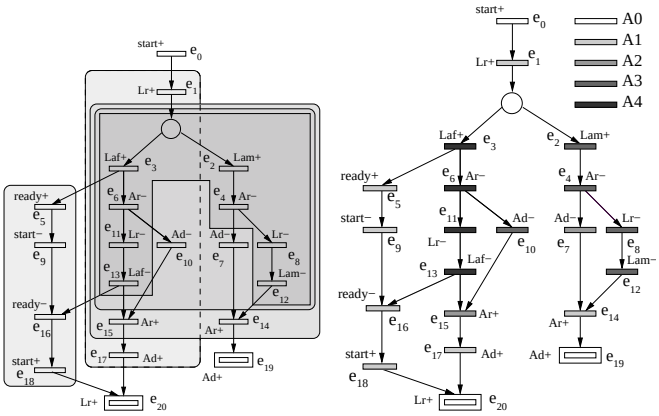
One can see that the implementations derived by signal insertion are more complex than the one obtained by concurrency reduction. These two implementations also delay signals $z$ and $x_1$, whereas the one derived using concurrency reduction does not introduce delays. Additionally, the solution obtained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

### B. A/D converter

The example shown in Fig. 8 is a part of the A/D converter proposed in [10]. It contains two type I and three type II cores shown in Fig. 8(a), and the corresponding height map is shown in Fig. 8(b). The valid transformations are presented in the tables in Fig. 8(c,d), which also show the total number of literals in the corresponding equations and the worst-case input-to-output delays. The equations for most interesting of these solutions are shown in Fig. 8(e).

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 $e_9$ are concurrent to the events in the peak and the concurrency starts in the peak. The first four solutions in the table in Fig. 8(c) 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). As a result, this is a very area-efficient solution — its literal count is just 11. The simulated worst-case input-to-output delay for this solution is 1.1ns ($Lam^+ \rightarrow Ar^- \rightarrow Lr^-$).




(a) unfolding prefix with cores (b) height map

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

| # | causal constraint | lits | delay |
|---|---|---|---|
| 1 | $h(e_9) \dashrightarrow h(e_{11})$ | 11 | 1.1ns |
| 2 | $h(e_5) \dashrightarrow h(e_{11})$ | 14 | 1.2ns |
| 3 | $h(e_5) \dashrightarrow h(e_6)$ | 14 | 1.5ns |
| 4 | $h(e_9) \dashrightarrow h(e_6)$ | 11 | 1.4ns |
| 5 | $h(e_{10}) \dashrightarrow h(e_{11})$ | n/a | n/a |

(c) concurrency reductions

| # | phase 1 | phase 2 | lits | delay |
|---|---|---|---|---|
| 6 | $\|\| e_3$ to $e_{11}$ | before $e_{16}$ | 16 | 1.2ns |
| 7 | after $e_3$ | before $e_{16}$ | 15 | 1.5ns |
| 8 | before $e_6$ | before $e_{16}$ | 16 | 1.2ns |
| 9 | before $e_{11}$ | before $e_{16}$ | 15 | 1.3ns |
| 10 | after $e_3$ | after $e_9$ | 18 | 1.5ns |
| 11 | after $e_3$ | $\|\| e_5$ to $e_{16}$ | 20 | 1.6ns |

(d) signal insertions

$$
\begin{aligned}
[\text{ready}] &= \text{Laf} \\
[\text{Lr}] &= \text{start} \cdot \text{Ad} \cdot \text{Ar} + \\
&\quad \text{Laf} \cdot (\text{Ar} + \text{start}) \\
[\text{Ar}] &= \overline{\text{Lam}} \cdot \overline{\text{Laf}} \cdot (\text{Ar} + \overline{\text{Ad}})
\end{aligned}
$$

equations for solution 1

$$
\begin{aligned}
[\text{ready}] &= \text{start} \cdot \text{ready} + \text{Laf} \\
[\text{Lr}] &= \text{start} \cdot \text{Ad} \cdot \overline{\text{ready}} \cdot \text{Ar} + \\
&\quad \text{Laf} \cdot (\text{Ar} + \overline{\text{ready}}) \\
[\text{Ar}] &= \overline{\text{Lam}} \cdot \overline{\text{Laf}} \cdot (\text{Ar} + \overline{\text{Ad}})
\end{aligned}
$$

equations for solution 2

$$
\begin{aligned}
[\text{ready}] &= \text{Laf} + \text{csc} \\
[\text{Lr}] &= \text{start} \cdot \text{Ad} \cdot \text{Ar} \cdot \overline{\text{csc}} + \\
&\quad \text{Laf} \cdot (\overline{\text{csc}} + \text{Ar}) \\
[\text{Ar}] &= \overline{\text{Lam}} \cdot \overline{\text{Laf}} \cdot (\text{Ar} + \overline{\text{Ad}}) \\
[\text{csc}] &= \text{start} \cdot \text{csc} + \text{Laf}
\end{aligned}
$$

equations for solution 6

$$
\begin{aligned}
[\text{ready}] &= \text{csc} \\
[\text{Lr}] &= \text{Ar} \cdot (\text{start} \cdot \overline{\text{csc}} \cdot \text{Ad} + \text{Laf}) \\
[\text{Ar}] &= \overline{\text{Lam}} \cdot \overline{\text{Laf}} \cdot (\text{Ar} + \overline{\text{Ad}}) + \\
&\quad \text{Laf} \cdot \overline{\text{csc}} \\
[\text{csc}] &= \text{start} \cdot \text{csc} + \text{Laf}
\end{aligned}
$$

equations for solution 7

$$
\begin{aligned}
[\text{ready}] &= \text{Laf} + \text{csc} \\
[\text{Lr}] &= \overline{\text{csc}} \cdot (\text{start} \cdot \text{Ar} \cdot \text{Ad} + \text{Laf}) \\
[\text{Ar}] &= \overline{\text{Lam}} \cdot \overline{\text{Laf}} \cdot (\text{Ar} + \overline{\text{Ad}}) \\
[\text{csc}] &= \text{start} \cdot \text{csc} + \text{Laf} \cdot \overline{\text{Ar}}
\end{aligned}
$$

equations for solution 9

(e) selected equations corresponding to valid transformations

Fig. 8. Top level of the A/D converter.

However, it is somewhat misleading, since it does not take into account that the introduced causal constraint indirectly delays $Laf^-$ by the input event $start^-$, which can be slow. The delay penalty estimate in out cost function better reflects the real situation.

The second solution does not delay $e_{11}$ but introduces an additional trigger to $Lr^-$. As a result, its literal count, 14, is larger than that for the first solution, but its delay estimate in the cost function is better. The simulated worst-case input-to-output delay for this solution is 1.2ns ($Lam^+ \rightarrow Ar^- \rightarrow Lr^-$). However, it is achieved not where the concurrency was reduced. In fact, the delays in the paths $Laf^+ \rightarrow ready^+ \rightarrow Lr^-$ and $Laf^+ \rightarrow Ar^- \rightarrow Lr^-$ are 1.0ns.

The third solution delays $e_6$ by $e_5$, and the fourth solution delays $e_6$ by $e_5$ and $e_9$; moreover, these two solutions introduce an additional trigger to *Ar* (which already had three triggers), and thus are inferior according to both our cost function and the simulation results.

Alternatively, the encoding conflicts can be solved by inserting a transition of a new signal *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 must not be delayed by newly inserted transitions, i.e., in the peak the transition of *csc* must not delay $e_3$ and $e_{13}$. Then 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 counterpart transition of *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 signal insertion solutions are presented in the table in Fig. 8(d).

Solution 6 introduces the smallest (among all signal insertions) delay (only $ready^-$ is delayed), and its literal count is 16. Its worst-case input-to-output delay is 1.2ns ($Lam^+ \rightarrow Ar^- \rightarrow Lr^-$), but it is achieved not in the branch where *csc* was inserted; the delays in the paths $start^- \rightarrow csc^- \rightarrow ready^-$ and $Laf^- \rightarrow csc^- \rightarrow ready^-$ are just 0.7ns.

Solution 7 has the smallest (among all signal insertions) estimated logic complexity, but quite a large delay (the insertion delays $ready^+$, $Ar^-$ and $ready^-$). Its literal count is 15 and the worst-case input-to-output delay is 1.5ns ($Laf^+ \rightarrow csc^+ \rightarrow Ar^- \rightarrow Lr^-$).

Solution 8 has the literal count 16 and the worst-case input-to-output delay of 1.2ns, which is as good as the delay of solution 6. However, unlike the worst-case delay in solution 6, it is achieved in the path $Laf^- \rightarrow csc^+ \rightarrow Ar^- \rightarrow Lr^-$, i.e., where *csc* was inserted.

Solutions 10 and 11 are inferior to other solutions in the table, which is in good agreement with our cost function. However, the literal count of solution 9 (which is also PETRIFY's solution) is 15, which is lower than our cost function suggests — in fact, it is equal to that of solution 7. This shows that our cost function is not perfect, since it uses quite a rough estimate of complexity, not taking the context signals into account. However, it worked well in the other cases and it is not trivial to significantly improve it without introducing a considerable time overhead. The worst-case input-to-output delay for this solution is 1.3ns ($Laf^+ \rightarrow Ar^- \rightarrow csc^+ \rightarrow Lr^-$).

Note that our combined framework explored quite a large design space and produced a wide range of solutions, allowing the designer to exploit the speed/area tradeoff and make an informed choice about which of them is the most appropriate for a given application (or allow the tool to chose the transformation using the cost function).

## V. CONCLUSIONS AND FUTURE WORK

This paper presents a combined framework for the resolution of encoding conflicts in STG unfoldings. It allows for exploring a larger design space and helps 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-self-concurrent Petri net [9].

The future work will be focused on the following issues:

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

*Acknowledgements:* The authors would like to thank Mark Josephs for answering our questions concerning the notion of refinement in [5] and Maciej Koutny, Walter Vogler and the anonymous reviewers for helpful comments. This research was supported by the Royal Academy of Engineering/EPSRC postdoctoral research fellowship EP/C53400X/1 (DAVAC) and the EPSRC grant GR/S12036 (STELLA).

## REFERENCES

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

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

[3] 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.

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

[5] M. Josephs: An Analysis of Determinacy Using a Trace-Theoretic Model of Asynchronous Circuits. Proc. of *ASYNC'03*, IEEE Comp. Soc. Press (2003) 121–131.

[6] V. Khomenko: *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD Thesis, School of Comp. Sci., Univ. of Newcastle (2003).

[7] V. Khomenko, M. Koutny and A. Yakovlev: Detecting State Coding Conflicts in STG Unfoldings Using SAT. *Fund. Inf.* 62(2) (2004) 1–21.

[8] V. Khomenko, M. Koutny and A. Yakovlev: Logic Synthesis for Asynchronous Circuits Based on Petri Net Unfoldings and Incremental SAT. *Fund. Inf.* 70(1-2) (2006) 49–73.

[9] V. Khomenko, A. Madalinski and A. Yakovlev: Resolution of Encoding Conflicts by Signal Insertion and Concurrency Reduction Based on STG Unfoldings. Tech. Rep. CS-TR-858, Univ. of Newcastle (2004).

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

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

[12] A. Madalinski, A. Bystrov, V. Khomenko and A. Yakovlev: Visualization and Resolution of Coding Conflicts in Asynchronous Circuit Design. *IEE Proceedings: Computers & Digital Techniques* 150(5) (2003) 285–293.

[13] V. Pratt: Modelling Concurrency with Partial Orders. *International Journal of Parallel Programming* 15(1) (1986) 33–71.

[14] H. Saito, A. Kondratyev, J. Cortadella, L. Lavagno and A. Yakovlev: What Is the Cost of Delay Insensitivity? Proc. of *CAD'99*, IEEE Comp. Soc. Press (1999) 316–323.

[15] M. Schäfer and W. Vogler: Component Refinement and CSC Solving for STG Decomposition. Tech. Rep. 2004-13, Institut für Informatik, Univ. Augsburg (2004).

[16] T. Verhoeff: Analyzing Specifications for Delay-Insensitive Circuits. Proc. of *ASYNC'98*, IEEE Comp. Soc. Press (1998) 172–183.