Efficient Automatic Resolution of Encoding Conflicts Using STG Unfoldings

Victor Khomenko

***Abstract*—Synthesis of asynchronous circuits from Signal Transition Graphs (STGs) involves resolution of state encoding conflicts by means of refining the STG specification. In this paper, a fully automatic technique for resolving such conflicts by means of insertion of new signals and concurrency reduction is proposed. It is based on *conflict cores*, i.e., sets of transitions causing encoding conflicts, which are represented at the level of finite and complete unfolding prefixes, and a SAT solver is used to find where in the STG the transitions of new signals should be inserted and to check the validity of concurrency reductions. The experimental results show significant improvements over the state space based approach in terms of runtime and memory consumption, as well as some improvements in the quality of the resulting circuits.**

***Index Terms*—Asynchronous circuits, encoding conflicts, concurrency reduction, STG, Petri net unfoldings, logic synthesis.**

I. Introduction

ASYNCHRONOUS circuits are a promising type of digital circuits. They have lower power consumption and electro-magnetic emission, no problems with clock skew and related subtle issues, and are fundamentally more tolerant of voltage, temperature and manufacturing process variations [1]. The International Technology Roadmap for Semiconductors report on Design [2] predicts that 22% of the designs will be driven by handshake clocking (i.e., asynchronous) in 2013, and this percentage will raise up to 40% in 2020.

PETRIFY [1] is one of the commonly used tools for synthesis of asynchronous circuits. As a specification it accepts a *Signal Transition Graph (STG)* [3], [4] — a class of interpreted Petri nets in which transitions are labelled with the rising and falling edges of circuit signals. For synthesis, PETRIFY employs the state space of the STG, and so it suffers from the combinatorial *state space explosion* problem. That is, even a relatively small system specification can (and often does) yield a very large state space. This puts practical bounds on the size of control circuits that can be synthesised using such techniques, which are often restrictive, especially if the specification is not constructed manually by a designer but rather generated automatically from high-level hardware descriptions. (For example, designing a control circuit with more than 20–30 signals with PETRIFY is often impossible.) Hence, this approach does not scale. Moreover, PETRIFY cannot guarantee a solution which can be mapped to the target gate library.

One way to cope with the state space explosion problem is to use *syntax-directed* translation of the specification to a circuit, avoiding thus building the state space. This is essentially the idea behind BALSA [5] and TANGRAM [6]. This technique, although computationally efficient, often yields circuits with large area and performance overheads compared with synchronous counterparts. This is because the resulting circuits are highly over-encoded, i.e., they contain many unnecessary state-holding elements.

For asynchronous circuits to be competitive, one has somehow to combine the advantages of logic synthesis (high quality of circuits) and syntax-directed translation (guarantee of a solution, efficiency) while compensating for their disadvantages. A natural way of doing this is to apply logic synthesis to the control path extracted from, e.g., a BALSA specification. This control path can be partitioned into smaller clusters which can be handled by logic synthesis, and the clusters on which it fails (because of either inability to find a solution in the target gate library or exceeding memory or time constraints) are implemented using the syntax-directed translation. The experiments conducted in [7] showed that such a combined approach can halve the area of control path and improve its latency, compared with the traditional syntax-directed translation, as long as clusters which can be confidently handled by logic synthesis are sufficiently large.

Arguably, one of the most difficult tasks in logic synthesis is resolution of *Complete State Coding (CSC)* conflicts, arising when semantically different (i.e., enabling different sets of outputs) reachable states of an STG have the same *encoding*, i.e., the binary vector representing the value of all the signals in a given state, as illustrated in Fig. 1(a,b). To resolve a CSC conflict, new *internal* signals helping to distinguish between these states must be inserted into the specification in such a way that its 'external' behaviour does not change. (Intuitively, insertion of a signal elongates the encoding, introducing thus additional memory into the circuit, helping to trace the current state.) The area and latency of the resulting circuit depend to a large extent on the way the new signals were inserted.

V. Khomenko is a Royal Academy of Engineering/EPSRC Post-Doctoral Research Fellow. He is affiliated with School of Computing Science, Newcastle University, UK. E-mail: Victor.Khomenko@ncl.ac.uk.

This research was supported by the Royal Academy of Engineering/EPSRC post-doctoral research fellowship EP/C53400X/1 (DAVAC).

The design flow advocated in [7] is as follows. Given a (potentially large) STG, the CSC conflicts are resolved using an integer linear programming (ILP) technique to approximate the state space of an STG. Then the resulting STG (free from CSC conflicts) is decomposed into smaller components in such a way that they are also free from CSC conflicts, as described in [8]. (Typically, each component is responsible for producing a single signal.) Then these components are synthesised one-by-one using PETRIFY. This approach can handle much larger specifications than PETRIFY alone, but its scalability is still limited since ILP is an NP-complete problem. For example, [7] reports that for the ART(20,9) benchmark with 436 places, 398 transitions and 199 signals it took over an hour to resolve CSC conflicts with area optimisation, and over two hours with delay optimisation. Moreover, an ILP approximation of the state space may work poorly for some STGs, e.g., those containing self-loops (i.e., pairs of arcs (p,t), (t,p) going in opposite directions).

In this paper, we follow a more scalable approach, which avoids performing expensive operations (such as resolving CSC conflicts) on the original STG. It works by proceeding with decomposition immediately, without resolving CSC conflicts. Hence, the resulting components, unlike ones in the technique described above, are not free from CSC conflicts. If a component has a CSC conflict, it can happen due to one of the following two reasons: (i) this conflict was present already in the original STG; or (ii) this conflict was introduced because some of the signals preventing it in the original STG are not present in the component. The technique described in [9] allows one to check which of these two reasons applies, and in case (ii) to find signals which need to be added to the component to prevent such CSC conflicts. Finally, the remaining CSC conflicts are resolved in each component, and the resulting STGs are synthesised.

Although this approach is quite scalable, it can be successful only if resolution of CSC conflicts and logic synthesis can be efficiently performed for all components, since a failure to synthesise even one of them means that the whole STG is not synthesised. In particular, PETRIFY may be inadequate for this task because of its rather restrictive limitations on the size of components. A more promising approach is to employ STG unfolding prefixes [10]–[12].

A *finite and complete unfolding prefix* of an STG is a finite acyclic net which implicitly represents all the reachable states of this STG together with transitions enabled at those states. Intuitively, it can be obtained through *unfolding* the STG, by successive firing of transitions, 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.

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 finite 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 *choices* (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). Intuitively, a configuration is a partially ordered execution, i.e., an execution where the order of firing of some of its events (viz. concurrent ones) is not important. We will denote by $[e]$ the *local* configuration of an event e, i.e., the smallest (w.r.t. $\subset$) configuration containing e (it is comprised of e and its causal predecessors).

The unfolding is infinite whenever the original STG has an infinite run; however, if the STG 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. Intuitively, an event e can be declared cut-off if the already built part of the prefix contains a configuration C^e (called the *corresponding* configuration of e) such that its final marking and encoding coincide with those of $[e]$ [13] and C^e is smaller than $[e]$ w.r.t. some well-founded partial order on the configurations of the unfolding, called an *adequate order* [10]. Fig. 1(c) shows a finite and complete unfolding prefix of the STG shown in Fig. 1(a); the only cut-off event is depicted as a double box, and its corresponding configuration is $\{e_1, e_2\}$.

Efficient algorithms exist for building such prefixes [10], [11], which ensure that the number of non-cut-off events in a complete prefix can never exceed the number of reachable states of the STG. Moreover, complete prefixes are often exponentially smaller than the corresponding state graphs, especially for highly concurrent STGs, because they represent concurrency directly rather than by multidimensional 'diamonds' as it is done in state graphs. For example, if the original STG 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 [14] they are just slightly bigger than the original STGs themselves. Thus, unfolding prefixes are well-suited for alleviating the state space explosion.

In [14] the unfolding technique was applied to detection of CSC conflicts between reachable states of an STG. Moreover, in [15] the problem of complex-gate logic synthesis from an STG free from CSC conflicts was solved. The experiments in [14], [15] showed that unfolding-based approach can handle much bigger STGs then PETRIFY.

The visualisation method presented in [16] is aimed at facilitating a manual refinement of an STG with CSC conflicts, and works on the level of unfolding prefixes. In order to avoid the explicit enumeration of CSC conflicts, they are visualised as *cores*, i.e., sets of transitions 'causing' one or more of them.

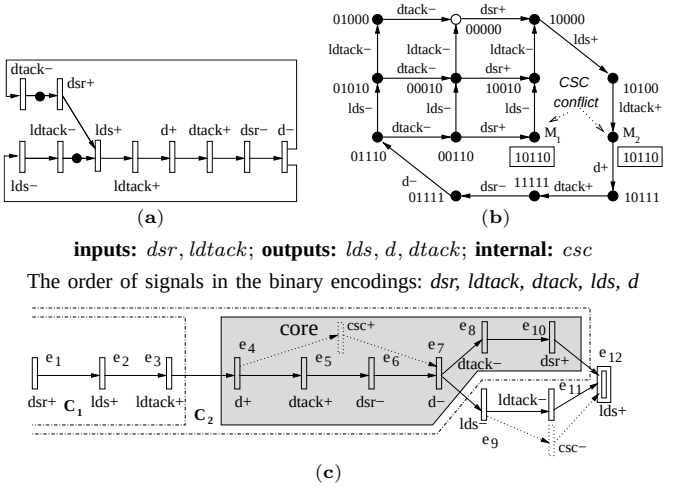


Fig. 1. An STG modelling the read cycle of the VME bus controller **(a)**, its state graph showing a CSC conflict between the states M_1 and M_2 **(b)**, and its unfolding prefix showing the conflict core corresponding to this CSC conflict and a way to resolve it by insertion of a new internal signal csc **(c)**.

(A core can be computed as the symmetric set difference of two configurations whose final states are in CSC conflict.) All such cores must eventually be eliminated, e.g., by adding new internal signals that resolve the CSC conflicts, to yield an STG satisfying the CSC property. This approach is illustrated in Fig. 1(c). One can see that the encodings at the beginning and at the end of the core are the same. This suggests that a core can be eliminated by the introduction of a new signal, csc, in such a way that one of its transitions is inserted into the core, as this would violate the stated property. Note that at least two transitions, viz. the falling and the rising edges of the signal, have to be inserted into the STG in order to preserve the *consistency* [1], [3] — a necessary condition for implementability of an STG as a circuit, ensuring that all the state encodings are binary; in particular, for every signal s, the following two properties must hold: (i) in all executions of the STG, the first occurrence of a transition of s has the same sign (either rising of falling); (ii) the rising and falling transitions of s alternate in every execution. In this example, the new transitions were inserted concurrently to existing ones in order to minimise the latency of the circuit. After transferring them into the STG, no more CSC conflicts remain in it, and so one can proceed with logic synthesis. (Other ways of inserting a signal in this example are also possible — see Section V.)

The semi-automatic approach of [16] is only feasible for synthesis of small 'handcrafted' blocks. In this paper, we present a technique which is also based on cores in the STG unfolding prefix, but is *fully automatic* and can handle much larger STGs than PETRIFY, while delivering high-quality circuits. Together with [14], [15], [17], it essentially completes the design cycle for synthesis of asynchronous circuits from STGs that does not involve building reachability graphs at any stage and yet is a fully fledged logic synthesis. The conducted experiments show that the proposed method has significant advantage both in memory consumption and in runtime compared with the existing state space based methods, while delivering somewhat better circuits compared with those produced PETRIFY and the ILP method of [7]. Combined with the decomposition approach of [9], this design cycle can be applied for control re-synthesis of BALSA or TANGRAM specifications as described above.

This is the full version of the conference paper [18], with an additional contribution describing resolution of encoding conflicts using concurrency reduction (Section VI).

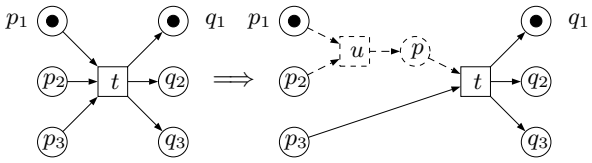
II. TRANSFORMATIONS

In this paper, we are primarily interested in *SB-preserving* transition insertions, i.e., ones preserving safeness and behaviour of the STG (in the sense that the original and the transformed STGs are weakly bisimilar, provided that the newly inserted transitions are considered silent). Below we describe several kinds of transition insertions, which will be used for CSC conflict resolution, and the algorithms presented in [17] allow one to check their validity.

We assume that the original STG is *input-proper*, i.e., no transition of an internal signal can trigger a transition of an input signal (as this is not implementable in a speed-independent way). All the transformations used for resolution of encoding conflicts in this paper preserve this property.

Building an unfolding prefix of an STG can be a time-consuming operation. However, in most practical cases the approach described in [17] allows one to avoid a potentially expensive re-unfolding after each transition insertion, by performing local modifications in the existing prefix instead. Moreover, it yields a prefix similar to the original one, which is advantageous for visualisation and allows one to transfer some information (e.g., the yet unresolved CSC cores) from the original prefix to the modified one.

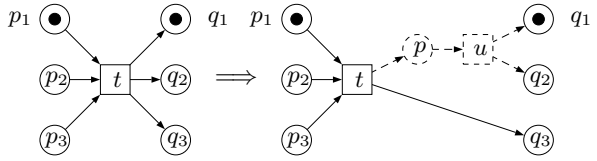
Sequential pre-insertion: A sequential pre-insertion is essentially a generalised transition splitting, and is defined as follows. Given a transition t and a set of places $S \subseteq {}^\bullet t$, the sequential pre-insertion $S \wr t$ is the transformation inserting a new transition u (with an additional place) 'splitting off' the places in S from t. The picture below illustrates the sequential pre-insertion $\{p_1, p_2\} \wr t$.



One can easily show that sequential pre-insertions always preserve safeness and traces (i.e., firing sequences with the silent transitions removed). However, in general, the behaviour is not preserved, and so a sequential pre-insertion is not guaranteed to be SB-preserving (in fact, it can introduce deadlocks) [17]. Given an unfolding prefix, it is quite easy to check whether a pre-insertion is SB-preserving [17].

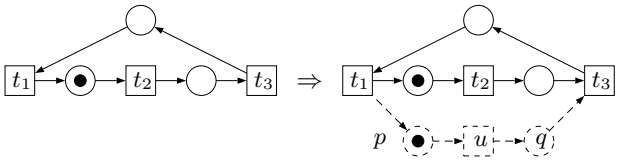
If a sequential pre-insertion $S \wr t$ is applied to an STG, the inserted transition should not 'delay' an input (as this would impose a constraint on the environment which was not present in the original specification), and so t must be a non-input transition. Moreover, one should take care that the *output-persistency* (i.e., the property that an enabled output cannot be disabled by another transition) is not violated; [17] presents an algorithm for checking that the newly inserted transition is not in a dynamic choice relation with any other transition, which ensures output-persistency preservation.

Sequential post-insertion: Similarly to sequential pre-insertion, sequential post-insertion is also a generalisation of transition splitting, and is defined as follows. Given a transition t and a set of places $S \subseteq t^\bullet$, the sequential post-insertion $t \wr S$ is the transformation inserting a new transition u (with an additional place) 'splitting off' the places in S from t. The picture below illustrates the sequential post-insertion $t \wr \{q_1, q_2\}$.



One can easily show that sequential post-insertions are always SB-preserving, and, when applied to an STG, preserve output-persistency. However, one still has to ensure that the inserted transition does not 'delay' any input transitions.

Concurrent insertion: Concurrent transition insertion can be advantageous for performance, since the inserted transition can fire in parallel with the existing ones. It is defined as follows. Given two distinct transitions, t' and t'', and an $n \in \{0, 1\}$, the concurrent insertion $t' \overset{n}{\dashv\!\!\rightarrow} t''$ is the transformation inserting a new transition u (with a couple of additional places) between t' and t'', and putting n tokens in the place in its preset. We will write $t' \dashv\!\!\rightarrow t''$ instead of $t' \overset{0}{\dashv\!\!\rightarrow} t''$ and $t' \overset{\bullet}{\dashv\!\!\rightarrow} t''$ instead of $t' \overset{1}{\dashv\!\!\rightarrow} t''$. The picture below illustrates the concurrent insertion $t_1 \overset{\bullet}{\dashv\!\!\rightarrow} t_3$ (note that the token in p is needed to prevent a deadlock).



In general, concurrent insertions preserve neither safeness nor behaviour. In [17], an efficient test whether a concurrent insertion is SB-preserving, working on an unfolding prefix, has been developed.

If a concurrent insertion $t' \overset{n}{\dashv\!\!\rightarrow} t''$ is applied to the STG, the output-persistency is guaranteed to be preserved, but the inserted transition should not 'delay' an input, and so t'' must be a non-input transition.

Equivalent transformations: It can happen that a sequential post-insertion $t \wr S$ yields essentially the same net as a sequential pre-insertion $S' \wr t'$, where $t \in {}^{\bullet\bullet}t'$; in particular, this happens if $S \cup S' \subseteq t^\bullet \cap {}^\bullet t'$ and $|{}^\bullet p| = |p^\bullet| = 1$ for all $p \in S \cup S'$. In such a case there is no reason to distinguish between these two transformations, e.g., one can convert a post-insertion into an equivalent pre-insertion whenever possible. Moreover, since post-insertions are always SB-preserving, there is no need to check the validity of the resulting transformation.

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

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

It is important to note that an SB-preserving transition insertion remains SB-preserving if another commuting SB-preserving transition insertion is applied first. Hence transformations whose validity has been checked can be cached, and after some transformation has been applied, the non-commuting transformations are removed from the cache and the new transformations that became possible in the modified STG are computed, checked for validity and added to the cache. (In particular, in the proposed CSC conflict resolution procedure, there is no need to check the validity of a particular transformation if it was checked in a preceding iteration.)

A *composite* transition insertion is a transformation defined as the composition of several pairwise commutative transition insertions. Clearly, if a composite transition insertion consists of SB-preserving transition insertions then it is SB-preserving, i.e., one can freely combine SB-preserving transition insertions, as long as they are pairwise commutative. This property is useful for conflict resolution: typically, several transitions of a new internal signal have to be inserted in each iteration of the algorithm, in order to preserve the consistency of the STG. For example, in Fig. 1(c) a composite transformation comprising two commuting SB-preserving concurrent insertions (adding the new transitions csc^+ and csc^-) has been applied in order to resolve the CSC conflict while preserving the consistency of the STG. (Note that the transformation is applied to the STG, and then is reflected in the prefix, without re-unfolding.)

III. Resolution of CSC conflicts

On each iteration of the proposed CSC conflict resolution procedure, a consistency-preserving composite insertion $\mathcal{I}$ resolving some of the conflict cores is chosen.

Given a finite and complete prefix of the STG unfolding, one can compute a set $\mathfrak{I}$ of *valid* (i.e., SB-preserving, output-persistency-preserving, not delaying an input, etc.) insertions as described in the previous section. (There is only a polynomial in the size of the STG number of such insertions if $\max\bigcup_{t\in T}\{|{}^\bullet t|, |t^\bullet|\}$ is bounded by a constant, as the number of sequential insertions is then linear in the number of STG's transitions, since for each t the number of insertions of the form $t \wr S$ or $S \wr t$ is bounded by a constant, and the number of concurrent insertions is quadratic in the number of STG's transitions.) Then we formulate a SAT problem as follows.

For each insertion $I \in \mathfrak{I}$ we create a Boolean variable, also denoted by I, indicating whether $I \in \mathcal{I}$. The constraints below ensure that for any satisfying assignment of a SAT instance to be built, the corresponding composite insertion $\mathcal{I}$ (obtained by taking the insertions whose corresponding variables are assigned 1) is *valid* (i.e., that it preserves the consistency of the STG, the chosen individual insertions commute, are not in the choice relation, and cannot trigger one another) and that some of the conflict cores are resolved (i.e., some progress is made). This SAT instance will be the conjunction of the constraints described below.

$\mathcal{MUTEX}$ constraint

Two signal insertions, I and I', are called *mutually exclusive* if they are non-commuting, or the inserted transitions are either concurrent or in the choice relation or can trigger one another.

All these conditions can be checked statically on the prefix (i.e., they are not encoded as a part of the Boolean formula), and one can build an undirected graph $\mathcal{G}$ representing the 'mutually exclusive' relation on $\mathfrak{I}$. Then, for every edge $\{I, I'\}$ of $\mathcal{G}$, the transformations I and I' must not be used together, which is expressed by the constraint:

$$\bigwedge_{\{I,I'\}\in edges(\mathcal{G})} (\neg I \vee \neg I') \ .$$

The size of this constraint can be quadratic in $|\mathfrak{I}|$. A smaller translation can be obtained by heuristically covering the edges of $\mathcal{G}$ by minimum number of cliques (using, e.g., the heuristic algorithm described in [20]), trying also to minimise the sizes of individual cliques, and generating the constraint $\sum_{I\in Cl} I \leq 1$ for each clique Cl. A linear in $|Cl|$ translation of this pseudo-Boolean constraint into a Boolean formula is possible by introducing auxiliary variables [21], [22].

Sign alternation constraint

The chosen SAT encoding does not carry any information concerning the signs ('+' or '−') of the inserted transitions. This is motivated by the desire to reduce the number of variables in the corresponding SAT instance by exploiting the following symmetry: it is always possible to flip the signs of all the transitions corresponding to a given internal signal without affecting the correctness (consistency, output-persistency, etc.) of the STG. However, one still has to ensure that consistent assignment of signs to each signal insertion within the composite signal insertion is possible; given such a composite insertion, one can statically compute the assignment using a prefix, by arbitrarily choosing the initial value (0 or 1) of the newly inserted signal. Hence, without loss of generality, one can assume that this value is 0 (it can be easily changed to 1 by flipping the signs of all the transitions corresponding to the newly inserted signal after the CSC conflict resolution process is completed).

In part, this condition is ensured by the $\mathcal{MUTEX}$ constraint, which guarantees that the instances of the newly inserted signals are not concurrent, and so within any configuration they are totally ordered w.r.t. the causality relation. The purpose of the sign alternation constraint $\mathcal{SA}$ is to ensure that the signs of the instances of the newly inserted signal alternate in each configuration of the prefix.

Given a configuration C of the prefix and a composite insertion $\mathcal{I}$, we denote by $Code_{\mathcal{I}}(C)$ the encoding of the newly inserted signal at the final state of C. (Recall that we assume that the initial value of this signal is 0, i.e., $Code_{\mathcal{I}}(\emptyset) \stackrel{\text{df}}{=} 0$.)

Let $J_0, \ldots, J_k$ be the instances of I in the prefix, i.e., the I-labelled events which would be added to the prefix if the insertion I is applied to the STG. (They can be computed statically on the prefix [17].) We extend the usual notation for presets and postsets to transformation instances; but note that, depending on the type of insertion, ${}^\bullet J_i$ or $J_i^\bullet$ (or both) may be not in the prefix (until the transformation is applied). However, the events in ${}^{\bullet\bullet}J_i$ are in the prefix even before the transformation is applied.

For a configuration C, let $\#_I C$ be the number of instances of I which would be inserted by the transformation I into C; it can be computed statically as follows:

$$\#_I C \stackrel{\text{df}}{=} \begin{cases} \#_{t''} C & \text{if } I \text{ is } t' \stackrel{n}{\longmapsto} t'' \\ \#_t C & \text{if } I \text{ is } S \wr t \\ \#_t C - m & \text{if } I \text{ is } t \wr S, \end{cases}$$

where $\#_t C$ denotes the number of t-labelled events in C, and $m = 1$ if C can be extended by some instance of I and $m = 0$ otherwise (i.e., the 'hanging' instance of a sequential post-insertion I is not counted, as it is not inside the configuration).

Assuming that the instances of the new signal within C can be assigned signs in a consistent way, $Code_{\mathcal{I}}(C)$ can be expressed as follows:

$$Code_{\mathcal{I}}(C) \iff \bigoplus_{I\,:\,\#_I C \text{ is odd}} I \ .$$

(An auxiliary Boolean variable, also denoted $Code_{\mathcal{I}}(C)$, together with the above constraint defining its value, is introduced in the SAT instance being built if $Code_{\mathcal{I}}(C)$ appears in the formulae below.)

The sign alternation constraint $\mathcal{SA}$ needs to ensure that if $I \in \mathcal{I}$ then all its instances $J_0, \ldots, J_k$ can be assigned the same sign in a consistent way, i.e., that the values of

$Code_{\mathcal{I}}([^{\bullet\bullet}J_0]), \ldots, Code_{\mathcal{I}}([^{\bullet\bullet}J_k])$ are the same, where $[X]$ denotes the minimal (w.r.t. $\subset$) configuration containing all the events in X. This can be accomplished, for each $I \in \mathfrak{I}$, by the following constraint:

$$I \Rightarrow \mathcal{SAME}(Code_{\mathcal{I}}([^{\bullet\bullet}J_0]), \ldots, Code_{\mathcal{I}}([^{\bullet\bullet}J_k])) \ ,$$

where

$$\mathcal{SAME}(x_0, \ldots, x_k) \stackrel{\text{df}}{=} \bigwedge_{i=0}^{k} (x_i \Rightarrow x_{i+1 \bmod (k+1)}) \ .$$

Since for a given t, all insertions of the form either $t \wr \cdot$ and $t \stackrel{\bullet}{\vdash\!\!\!\rightarrow} \cdot$ or $t \stackrel{\bullet}{\vdash\!\!\!\rightarrow} \cdot$ have the same $^{\bullet\bullet}J_0, \ldots, ^{\bullet\bullet}J_k$, the sign alternation constraints for a group G of such insertions can be combined as follows:

$$\Big(\bigvee_{I \in G} I \Big) \Rightarrow \mathcal{SAME}(Code_{\mathcal{I}}([^{\bullet\bullet}J_0]), \ldots, Code_{\mathcal{I}}([^{\bullet\bullet}J_k])) \ .$$

Note that the $\mathcal{SA}$ constraint is defined via $Code_{\mathcal{I}}([^{\bullet\bullet}J])$ for all instances J of all the insertions $I \in \mathfrak{I}$, and the definition of $Code_{\mathcal{I}}(C)$ assumes that the instances of the new signal within C can be assigned signs in a consistent way, i.e., they are not concurrent (which is ensured by $\mathcal{MUTEX}$) and their signs alternate, which has to be ensured by $\mathcal{SA}$. This mutual dependency of $Code_{\mathcal{I}}(C)$ and $\mathcal{SA}$ does not cause problems, though, due to the following inductive argument. Suppose $\mathcal{SA}$ is incorrect for some configuration C of the prefix. Since $Code_{\mathcal{I}}(X)$ is computed correctly whenever $\mathcal{SA}$ is correct on X, and due to $\mathcal{MUTEX}$ no two instances of the new signal can be concurrent, $\mathcal{SA}$ must be incorrect already for the configuration $[^{\bullet\bullet}J] \subset C$ for some instance J of $I \in \mathfrak{I}$. Since $\subset$ is a well-founded order and $\mathcal{SA}$ is correct for the empty configuration, we have a contradiction.

$\mathcal{CUTOFF}$ constraint

The sign alternation constraint ensures that the signs of instances of the newly inserted signal will alternate in any configuration of the prefix. However, to guarantee consistency, one still has to add a constraint $\mathcal{CUTOFF}$ ensuring that this is also the case for the configurations of the full unfolding *beyond the cut-off events* of the prefix. For this, it is enough to ensure for each cut-off event e that after $\mathcal{I}$ is applied, the value of the newly inserted signal is the same in the final states of $[e]$ and its cut-off corresponding configuration.

One may be tempted to express this constraint as

$$Code_{\mathcal{I}}([e]) \iff Code_{\mathcal{I}}(C^e) \ ,$$

for each cut-off event e with a corresponding configuration C^e. However, it does not take into account the following subtlety. It can happen that some instance J of a post-insertion $I \in \mathcal{I}$ is such that C^e can be extended by J. The definition of $Code_{\mathcal{I}}$ does not take J into account (since J will not be in C^e after the transformation is applied), even though it may become a part of the corresponding configuration of e after I is applied. To capture this, a post-insertion I is called *e/C^e-mismatching* if some instance J of I is such that C^e can be extended by J and $[e]$ cannot be extended by J. Now such additional instances of post-insertions can be taken into account as follows:

$$Code_{\mathcal{I}}([e]) \iff Code_{\mathcal{I}}(C^e) \oplus \bigoplus_{I \in \mathcal{MM}^e} I \ ,$$

for each cut-off event e with a corresponding configuration C^e, where $\mathcal{MM}^e$ is the set of e/C^e-mismatching post-insertions.

As an optimisation, this constraint can be represented as

$$\neg \Big(Code_{\mathcal{I}}([e]) \oplus Code_{\mathcal{I}}(C^e) \oplus \bigoplus_{I \in \mathcal{MM}^e} I \Big) \ ,$$

and $\oplus$-sums can be optimised, as described at the end of this section. Alternatively, one can observe that if two post-insertions are commutative and non-concurrent then no configuration can be extended by both of them. Hence at most one of the variables in $\bigoplus_{I \in \mathcal{MM}^e} I$ can be assigned 1, i.e., one can replace this sub-expression by $\bigvee_{I \in \mathcal{MM}^e} I$. This can improve the runtime of SAT solver and shorten the formula, and the $\oplus$-sums can still be optimised for $Code_{\mathcal{I}}([e]) \oplus Code_{\mathcal{I}}(C^e)$.

$\mathcal{CORE}$ constraint

To ensure progress, a constraint conveying that at least one of the conflict cores is resolved, is added. Let $\mathcal{C}$ be a core. A signal insertion I is called *hanging w.r.t. $\mathcal{C}$* if, after it is applied, some of its instances directly precedes or succeeds $\mathcal{C}$. A composite transition insertion $\mathcal{I}$ is *hanging w.r.t. $\mathcal{C}$* if some $I \in \mathcal{I}$ is hanging w.r.t. $\mathcal{C}$.

One can observe that if $\mathcal{I}$ is hanging w.r.t. $\mathcal{C}$ then $\mathcal{C}$ is not resolved by $\mathcal{I}$. In the transformed prefix, this core will resurface as a core $\mathcal{C}'$, as one can always ensure that the encodings at the beginning and at the end of $\mathcal{C}'$ coincide by adding, if needed, a hanging instance of $I \in \mathcal{I}$ to the core.

$\mathcal{C}$ is resolved by a composite signal insertion $\mathcal{I}$ if an odd number of signal instances is inserted into it, and none of the inserted signal instances is hanging w.r.t. $\mathcal{C}$. By introducing new auxiliary variables $Hanging_{\mathcal{C}}$ and $Resolved_{\mathcal{C}}$ for each core $\mathcal{C}$, the $\mathcal{CORE}$ constraint is defined as follows:

$$\Big(\bigvee_{\mathcal{C}} Resolved_{\mathcal{C}} \Big) \wedge \bigwedge_{\mathcal{C}} \Big(Hanging_{\mathcal{C}} \iff \bigvee_{I \in H_{\mathcal{C}}} I \Big) \wedge$$

$$\bigwedge_{\mathcal{C}} \Big(Resolved_{\mathcal{C}} \iff \Big(\neg Hanging_{\mathcal{C}} \wedge \bigoplus_{\substack{I \notin H_{\mathcal{C}} \wedge \\ \#_I \mathcal{C} \text{ is odd}}} I \Big) \Big) \ ,$$

where $H_{\mathcal{C}}$ is the set of hanging w.r.t. $\mathcal{C}$ transition insertions.

Computation of $\oplus$-sums

One can notice that the constructed formulae contain many $\oplus$-sums over the same set of variables $\mathfrak{I}$. There is typically a lot of sharing between them, and so these sums can be optimised by computing common sub-sums only once.

The problem can be abstractly formulated as follows. Given m $\oplus$-sums over the variables $x_1, \ldots, x_n$, build a

small acyclic Boolean circuit[1] with n inputs and m outputs computing these $\oplus$-sums. (Such a circuit can then be converted into a Boolean formula in the conjunctive normal form, whose size is linear in the size of the circuit.)

This problem can be solved in a number of ways. The method described in [21, Chapter 4.7], [23] divides the variables into $n/\log n$ groups of $\log n$ variables each, computes all the possible sums in each group, and forms the circuit from these sums. For this, at most $\frac{n^2+mn}{\log n} - m$ binary $\oplus$-gates are needed. In the actual implementation, a method based on *preset trees* [11, Chapter 4] was used. Experiments show that it works quite well in practice.

Cost function

On each iteration of the method, a heuristic cost function is used to guide the search towards 'good' solutions with small area and/or performance overhead. The constructed SAT instance is solved several times, with constraints on the value of the cost function appended to the formula, so that a solution minimising the value of the cost function is eventually computed. (The process resembles a binary search on the value of the cost function.) The cost function we used is a weighted sum of the following components:

- the estimated number of unresolved CSC cores;
- the estimated number of unresolved *Universal State Coding (USC)* cores, i.e., cores corresponding to different states which have the same encoding (though USC cores which are not CSC cores are not harmful, they can turn into CSC cores once new signals are added);
- the estimated delay introduced by the insertion;
- the total number of syntactic triggers of all output and internal signals;
- the number of inserted transitions of a signal;
- the number of input signals which are not 'locked'[2] with the newly inserted signal;
- the number of output and internal signals which are not 'locked' with the newly inserted signal.

The user can choose the relative weights of the components of the cost function to guide the resolution process towards solutions with the desired area/latency trade-off. More details can be found in the technical report [19].

IV. Comparison with other techniques

In this section, the proposed technique for resolving CSC conflicts is compared with two other techniques: the one implemented in PETRIFY [1] and employing the state graphs, and the Integer Linear Programming (ILP) technique of [7].

[1]This Boolean circuit is an abstract construction needed for building a part of the SAT instance, and should not be confused with the circuit being synthesised from the STG.

[2]Two signals are in the 'lock' relation [24] if their instances (i) cannot be concurrent, and (ii) alternate in every execution sequence. 'Locking' the newly inserted signal with as many other signals as possible is a good heuristics for area optimisation [7].

PETRIFY's approach is well-documented in [1]. It works with state graphs, and thus does not scale. However, for small specifications it typically yields quite good solutions. Moreover, it has some additional capabilities which neither the ILP approach of [7] nor the proposed method have, viz. it can restructure the specification using net synthesis from the state graph. However, in practice the scalability is usually much more desirable than the ability to do restructuring (as it is useful only in very special cases).

The approach described in [7] works in a very different way. Instead of exact computation of the state space, it uses an approximate technique based on Integer Linear Programming (ILP). Briefly, this approach takes as an input a lasso-shaped CSC violation trace starting from the initial state and such that the two states, say, s_1 and s_2, in CSC conflict are positioned on the loop of the lasso. Then it tries to insert a set of new transitions (obtained by splitting existing transitions) corresponding to a new signal into the STG, in such a way that the STG remains consistent and the numbers of such transitions on the parts of the loop between s_1 and s_2, as well as between s_2 and s_1, are odd (i.e., the CSC conflict is resolved). For this, an ILP problem is formulated, whose solution gives a set of transitions which should be split (an elegant sufficient condition for the consistency of the resulting STG based on a place redundancy test is employed). Moreover, a heuristic cost function is used to guide the search towards solutions corresponding to circuits with either small area or small latency. This procedure is iterated until all the CSC conflicts are resolved.

The approach presented in this paper was inspired by that in [7], but it has a number of important differences. It iteratively inserts new internal signals into the specification until no CSC conflicts remain. On each iteration, it tries to eliminate some of the CSC conflict cores in the unfolding prefix [16] by insertion of new signals, guided by a heuristic cost function. The technique described in [17] is used to avoid re-unfolding the specification after each iteration, and to transfer the unresolved conflict cores from iteration to iteration. The main differences from [7] are described below.

- We use STG unfolding prefixes rather than STGs. This allows for an exact test of consistency where [7] used an approximate one, based on redundancy of places.
- We use a SAT rather than ILP solver. Besides, the SAT encoding of the problem is based on entirely different ideas.
- Unlike [7], the proposed method does not require a lasso-shaped CSC violation trace (in general, it is not always possible to find such a trace even if there are CSC conflicts), and uses a set of encoding conflict cores instead.
- Using unfoldings allows for efficient computation of violation traces using the technique described in [14]. In contrast, for methods working on the STG level, like that in [7], this is only possible for some restricted net classes, such as marked graphs or live and safe free-choice nets. Intuitively,

the problem of checking whether a given safe STG has CSC conflicts is PSPACE-complete [25, Proposition 5.1], while ILP is in NP, so the knowledge of a *Parikh vector* of the violation trace (i.e., a vector of non-negative numbers representing the number of times each transition fired in a given execution; it is typically returned by ILP methods [26]) does not help much — the reachability problem remains PSPACE-complete even if such a Parikh vector is provided as a part of the input. In principle, [7] could also use, e.g., the unfolding based technique of [14] for computing violation traces, but this would, to some degree, defeat the rationale of their approach, since an unfolding prefix of the STG has to be built for this — but then it would be natural to employ it for conflict resolution as well.

The actual approach used in [7] for computing a CSC violation trace works as follows [27] (unfortunately, this question was not addressed in [7]). The problem of CSC conflict detection is formulated as an ILP problem, which, if infeasible, guarantees that STG has no CSC conflicts. Otherwise, a Parikh vector of a CSC violation trace is computed, and an attempt is made to restore a trace from this Parikh vector by firing one-by-one the transitions corresponding to its non-zero components (the corresponding component of the Parikh vector is decremented after each firing). If at some point none of such transitions is enabled, one of them is anyway chosen and fired (leading to a 'negative' marking). The process stops when all the components of the vector become zero.

One can see that a violation trace is produced (and then resolved by insertion of new signals) even if the computed solution of the ILP problem is spurious (i.e., the corresponding CSC conflict states are unreachable). Moreover, the produced violation trace can be spurious (i.e., passing via negative markings) even if there is a real execution corresponding to the computed Parikh vector. Hence, the method of [7] can sometimes insert redundant signals resolving 'spurious' CSC conflicts.

- The transformations used in [7] were limited to simple transition splitting. The proposed approach allows one to use a much wider class of transformations; in particular, concurrent insertion and insertions splitting off just a part of a transition's preset or postset are possible.
- The proposed method takes into account multiple conflict cores, whereas the ILP approach considers only a single (perhaps, spurious) violation trace. In particular, this makes it possible to choose insertions which resolve many cores with one signal, reducing thus the total number of inserted signals and allowing for quicker progress — see the 8-way sequencer case study in Section V.
- Though the proposed approach is fully automatic, it inherits the visualisation possibilities described in [16], which may be useful for interaction with the user.

The described advantages come at the price of increasing the runtime compared with the method of [7]. However, the proposed method is much faster then PETRIFY and can handle quite large specifications. As it is intended for use in conjunction with the decomposition approach of [9], it fits well with practical applications such as control re-synthesis.

	Composite transition insertion	Lits
1	$\wr d^-, \quad \wr lds^+$	8
2	$\wr d^-, \quad ldtack^- \wr$	9
3	$\wr lds^+, \quad dtack^+ \mathbin{+\!\!\!\rightarrow} d^-$	11
4	$ldtack^- \wr, \quad dtack^+ \mathbin{+\!\!\!\rightarrow} d^-$	11
5	$d^+ \mathbin{+\!\!\!\rightarrow} d^-, \quad \wr lds^+$	11
6	$d^+ \mathbin{+\!\!\!\rightarrow} d^-, \quad ldtack^- \wr$	11
7	$\wr lds^+, \quad \wr dtack^+$	12
8	$ldtack^- \wr, \quad \wr dtack^+$	12
9	$\wr d^-, \quad lds^- \mathbin{\bullet\!\!+\!\!\!\rightarrow} lds^+$	12
10	$\wr lds^+, \quad dsr^- \mathbin{+\!\!\!\rightarrow} dtack^-$	12
11	$\wr lds^+, \quad \wr dtack^-$	13
12	$lds^- \mathbin{\bullet\!\!+\!\!\!\rightarrow} lds^+, \quad dtack^+ \mathbin{+\!\!\!\rightarrow} d^-$	14
13	$\wr lds^+, \quad dtack^+ \mathbin{+\!\!\!\rightarrow} dtack^-$	14
14	$d^+ \mathbin{+\!\!\!\rightarrow} d^-, \quad lds^- \mathbin{\bullet\!\!+\!\!\!\rightarrow} lds^+$	14
15	$d^+ \mathbin{+\!\!\!\rightarrow} dtack^-, \quad \wr lds^+$	14
16	$lds^- \mathbin{\bullet\!\!+\!\!\!\rightarrow} lds^+, \quad \wr dtack^+$	15
17	$\wr d^-, \quad \wr lds^+, \quad \wr dtack^+, \quad \wr dtack^-$	18

TABLE I
THE COMPOSITE TRANSITION INSERTIONS RESOLVING THE CSC CONFLICT SHOWN IN FIG. 1.

V. CASE STUDIES AND EXPERIMENTAL RESULTS

The CSC conflict resolution method described in this paper has been implemented in the MPSAT tool. In this section we present a number of case studies demonstrating some interesting features of the proposed approach, as well as the results of running it on a number of benchmarks. To solve the arising SAT instances, the MINISAT2 solver[3] has been used. All the experiments were conducted on a PC with a $Pentium^{TM}$ IV/3.4GHz processor and 2G RAM.

VME bus controller.

The specification of the read cycle of VME bus controller is shown in Fig. 1. Although it is a very small benchmark containing a single conflict core, MPSAT was able to find 17 possible ways to resolve it, listed in Table I. This shows that the proposed method explores a fairly large design space, including quite an unintuitive solution 17 with two set and two reset transitions, which resolves the core by inserting three transitions of csc into it. Many of these solutions cannot be computed by the method of [7], as the class of transformations it uses is limited to transition splitting.

An 8-way sequencer.

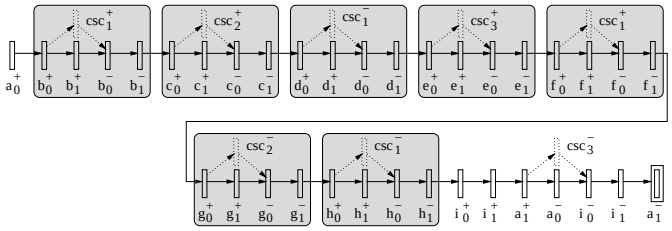
Sequencers are among the standard 'building blocks' of circuits produced from hardware description languages like

[3]Available from `www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/Main.html`.

BALSA and TANGRAM. The 'parent' handshake at port a initiates eight sequentially ordered 'child' handshakes at ports $b, \ldots, i$. Then the parent handshake completes, and the cycle continues. (The completion of the last 'child' handshake is reshuffled with the completion of the 'parent' handshake for an early acknowledgement at port a.) Fig. 2 shows the unfolding prefix of the STG specifying an 8-way sequencer with seven conflict cores.

Intuitively, at least three bits of additional memory are needed to implement this specification (by counting how many of the eight 'child' handshakes have been executed so far), so the CSC conflicts cannot be resolved by insertion of fewer than three signals. However, it is not trivial to find a solution using only three additional signals — in fact, PETRIFY's solutions has four new signals. MPSAT was able to find a fully concurrent solution with three signals shown in Fig. 2 by dotted lines. Note that to accomplish this the signal csc_1 is set and reset twice in each cycle.

Finding a solution with three signals is only possible by analysing multiple cores; the method of [7] cannot find such a solution because it analyses just a single violation trace on each iteration — in fact, it needed four signals to resolve the CSC conflicts in this case study.



inputs: $a_0, b_1, c_1, d_1, e_1, f_1, g_1, h_1, i_1$
outputs: $a_1, b_0, c_0, d_0, e_0, f_0, g_0, h_0, i_0$
internal: csc_1, csc_2, csc_3

Fig. 2. The unfolding prefix of an STG modelling an 8-way sequencer, showing 7 cores and a fully concurrent solution with 3 new signals.

Assorted small benchmarks

Table II compares the three methods for resolving CSC conflicts: the state-space based approach implemented in PETRIFY, the ILP approach of [7] (with post-processing removing redundant signals) and the one proposed in this paper, on a number of assorted small benchmarks from [7]. The meaning of the columns in the table is as follows (from left to right): the name of the problem; the number of places, transitions, and input and output signals in the original STG; the number of signals inserted by PETRIFY, the ILP approach of [7] and the approach proposed in this paper; and the number of literals in the final complex-gate implementations produced by the three approaches (the smallest numbers are highlighted). The numbers in the 'Pfy' and 'ILP' columns are as reported in [7], and, for consistency with [7], PETRIFY was used to synthesise the STGs after the CSC conflicts were resolved.

Example	STG $\|P\|/\|T\|$	*In/Out*	Signals Pfy	ILP	SAT	Literals Pfy	ILP	SAT
ADFAST	15/12	3/3	**2**	**2**	**2**	**14**	19	21
IRCV-BM	55/46	5/4	2	3	**1**	38	43	**28**
MMU	20/16	4/4	**3**	**3**	**3**	29	**26**	27
MMU0	20/16	4/4	**3**	**3**	**3**	29	**27**	**27**
MMU1	24/16	4/4	3	**2**	**2**	32	**23**	25
MR0	31/22	5/6	**3**	**3**	**3**	45	30	**29**
MR1	25/18	4/5	4	**3**	**3**	35	29	**27**
NAK-PA	22/18	4/5	**1**	**1**	**1**	**18**	**18**	**18**
NOWICK	19/14	3/2	**1**	**1**	**1**	**14**	**14**	**14**
PAR(4)	23/20	5/5	**4**	**4**	**4**	**32**	**32**	**32**
SEQ(8)	36/36	9/9	4	4	**3**	47	**37**	44
TSEND-BM	45/39	5/4	2	3	**1**	39	44	**27**
ALLOC-OUTBOUND	17/18	4/3	**2**	**2**	**2**	16	16	**15**
DUPLICATOR	14/12	2/2	**2**	**2**	**2**	19	**13**	**13**
MOD4_COUNTER	16/16	1/2	**2**	3	**2**	**25**	28	**25**
RAM-READ-SBUF	26/20	5/5	**1**	**1**	**1**	**18**	**18**	19
SBUF-RAM-WRITE	29/20	5/5	**2**	**2**	**2**	**22**	32	29
SBUF-READ-CTL	14/12	2/4	**1**	**1**	**1**	**15**	**15**	**15**
MASTER_1882	38/26	6/7	**1**	**1**	**1**	**38**	**38**	39
TRCV-BM	53/44	5/4	**2**	3	**2**	36	44	**34**
SEQ_MIX	20/20	4/4	3	**2**	**2**	**20**	**20**	**20**
SPEC_SEQ(4)	20/20	5/5	3	**2**	**2**	20	**19**	20
Total			51	51	**44**	601	585	**548**

TABLE II
EXPERIMENTAL RESULTS: ASSORTED SMALL STGS.[4]

One can see that in all cases the number of inserted by MPSAT signals was smaller or the same compared with the other methods, and also it produced smaller implementations (about 8.8% improvement over PETRIFY).

Scalable benchmarks

We also compared the described method with PETRIFY (the ILP tool of [7] was not available from the authors) on two groups of scalable benchmarks modelling m pipelines weakly synchronised without arbitration (PPWK(m, n)) and with arbitration (PPARB(m, n)). They are the benchmarks from the corresponding series used in [14], with the latter series modified by 'factoring out' the arbiter into the environment to ensure output-persistency. In these two series of benchmarks all the signals except the arbiter's grants in PPARB(m, n) 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 handshakes (sometimes with an inverter attached to the output if the environment acts as an active port). Fig. 3 illustrates these two types of STGs.

The results for these two groups are summarised in Table III, where the meaning of the columns is the same as in Table II, except that the sizes of the corresponding finite and complete prefixes (in terms of the numbers of conditions and events) are given in the forth column and the runtimes (in seconds) are now reported in the last three columns (for MPSAT,

[4]Two different sets of weights in the cost function were used to produce the numbers in the two 'SAT' columns: in the former the cost function was aimed at minimising the number of inserted signals (the literals were not taken into account and not reported), whereas in the latter it was aimed at minimising the number of literals in the final implementation (the signals were not taken into account and not reported).

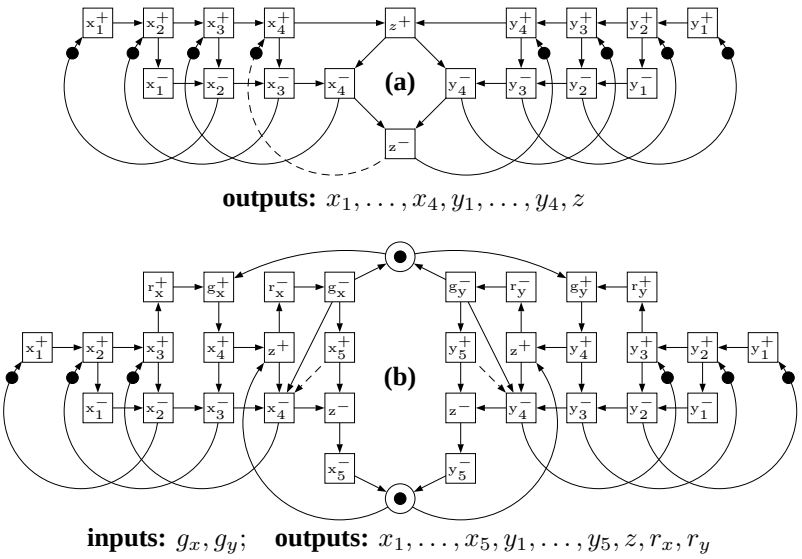


Fig. 3. STGs modelling two weakly synchronised pipelines without arbitration **(a)** and with arbitration **(b)**. The dashed arcs show how to resolve encoding conflicts using concurrency reduction.

Example	STG $\|P\|/\|T\|$	*In/Out*	Prefix $\|B\|/\|E\|$	Signals Pfy	SAT	Literals Pfy	SAT	Time, [s] Pfy	SAT^s	SAT^l
Marked Graphs										
PpWk(2,3)	23/14	0/7	41/24	1	1	30	29	<1	<1	<1
PpWk(2,6)	47/26	0/13	119/63	1	1	60	59	5	<1	<1
PpWk(2,9)	71/38	0/19	233/120	1	1	90	89	34	<1	6
PpWk(2,12)	95/50	0/25	383/195	1	1	120	119	368	<1	18
PpWk(3,3)	34/20	0/10	63/36	2	2	48	46	4	<1	<1
PpWk(3,6)	70/38	0/19	183/96	2	2	93	91	105	<1	6
PpWk(3,9)	106/56	0/28	357/183	2	2	138	136	1838	4	55
PpWk(3,12)	142/74	0/37	585/297	—	2	—	181	*mem*	5	175
STGs with Arbitration										
PpArb(2,3)	48/32	2/13	110/66	2	2	63	67	35	<1	2
PpArb(2,6)	72/44	2/19	218/120	2	3	93	97	118	1	17
PpArb(2,9)	96/56	2/25	362/192	2	2	123	127	1041	2	50
PpArb(2,12)	120/68	2/31	542/282	—	3	—	157	*mem*	8	159
PpArb(3,3)	71/48	3/19	188/114	3	3	100	105	620	<1	14
PpArb(3,6)	107/66	3/28	368/204	3	3	145	149	5043	2	117
PpArb(3,9)	143/84	3/37	602/321	3	4	190	194	12307	7	354
PpArb(3,12)	179/102	3/46	890/465	—	5	—	239	*mem*	24	839

TABLE III
EXPERIMENTAL RESULTS: SCALABLE PIPELINES.[4]

the runtimes for signal and literal optimisation are reported separately). We use *'mem'* if there was a memory overflow. It also should be noted that since PETRIFY was not able to synthesise some of the resulting STGs, they were synthesised with the unfolding-based technique described in [15], that is implemented in MPSAT.

One can see that on these benchmarks PETRIFY and MPSAT were very close in terms of the number of inserted signals and the number of literals. However, in terms of runtime and memory consumption MPSAT was clearly superior: in some cases the runtime differed by orders of magnitude, and the cases which were intractable for PETRIFY due to memory overflow were solved by MPSAT relatively easily.

It should be noted that, depending on whether signals or literals are minimised, MPSAT's runtimes can differ significantly on the same benchmark. This can be explained by the fact that in the former case many of the parameters of the cost function (viz. the estimated delay, the total number of syntactic triggers of all output and internal signals, the number of inserted transitions, the numbers of inputs and outputs which are not 'locked' with the newly inserted signal) are not taken into account (resulting in a considerable shortening of the SAT instance), whereas in the latter case only the estimated delay is not taken into account.

VI. RESOLUTION OF ENCODING CONFLICTS USING CONCURRENCY REDUCTION

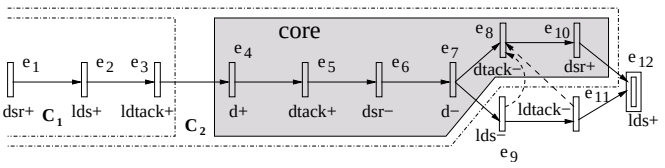


Fig. 4. VME bus controller: resolving the encoding conflict with the help of one of the concurrency reductions shown by the dashed arcs.

Another way of resolving the encoding conflict in the VME bus controller example is by eliminating the concurrency between either lds^- and $dtack^-$ or $ldtack^-$ and $dtack^-$, as shown by the dashed arcs in Fig. 4. These transformations 'drag' either lds^- or lds^- and $ldtack^-$ into the conflict core, destroying it. (In effect, state M_1 becomes unreachable, cf. Fig. 1(b)). The general ways of eliminating CSC conflict cores by 'dragging' existing events into the core are illustrated in Fig. 5(b,c) (see [28] for more details).

The former concurrency reduction yields an implementation with 10 literals, and the latter with only 7 literals, which

compares rather favourably with the implementations given in Table I, especially with fully concurrent ones. Of course, this comes at the price of sequentialising the STG, in particular the second concurrency reduction makes $dtack^-$ wait for an input transition, which might adversely affect the performance.

In practice it is often the case that concurrency reduction produces smaller circuits, which may also be faster due to simplification of the gates (even though the system manifests less concurrency, its events take less time to fire). Hence the common belief that more concurrency increases the performance is questionable in this context. 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, increasing thus 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, eliminating 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 [29], [30] or are restricted to specific net classes or use local transformations [31] and thus restrict the design space.

Formally, given an STG, a set of its transitions $U \neq \emptyset$, a transition $t \notin U$ and an $n \in \{0, 1\}$, a *concurrency reduction* $U \overset{n}{\dashrightarrow} t$ is defined as the transformation adding 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. 5(a). 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; in particular, no new traces are added (and thus the consistency is preserved).

Validity

Given a concurrency reduction $U \overset{n}{\dashrightarrow} t$ and a configuration C of the unfolding, we define $Tokens(C) \overset{\text{df}}{=} n + \#_U C - \#_t C$, where $\#_S C$ denotes the number of S-labelled (i.e., labelled by a transition in S) events in C, and $\#_t C \overset{\text{df}}{=} \#_{\{t\}} C$. Intuitively, $Tokens(C)$ is the final number of tokens in the newly inserted place (provided that C is a configuration of the unfolding of the modified Petri net as well), i.e., this is essentially the marking equation (see [32]) for this place. Note that $Tokens(C)$ can be negative.

In [28] a framework for unfolding-based resolution of encoding conflicts using concurrency reduction was developed. In particular, a notion of correctness of a concurrency reduction was proposed and justified. This notion is rather complicated (note that even language equivalence does not hold), and we do not present it in this paper. Instead, we give a slightly reformulated sufficient correctness condition proven in [28]. This condition assumes *weak fairness*, i.e., that a transition cannot remain enabled forever: it must either fire or be disabled by another transition firing.[5] 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.

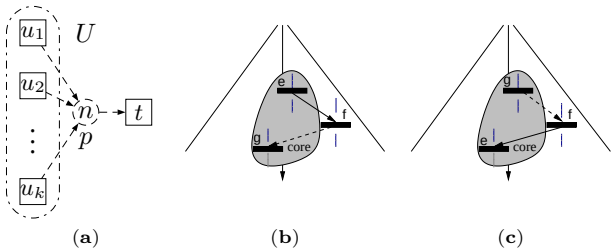


Fig. 5. Concurrency reduction $U \overset{n}{\dashrightarrow} t$ **(a)** and core elimination by concurrency reduction **(b,c)**.

In the proposition below, which is a slightly re-formulated version of [28, Proposition 3.2], we relax the definition of a configuration by allowing it to be infinite. A *maximal configuration* is a configuration which cannot be extended by another event. (Note that maximal configurations are either deadlocked or infinite, though not every infinite configuration is maximal.) We also define by $\langle e \rangle \overset{\text{df}}{=} [e] \setminus \{e\}$ the set of causal predecessors of an event e. Intuitively, this proposition states that a concurrency reduction $U \overset{n}{\dashrightarrow} t$ is valid if every maximal configuration Π of the unfolding of the STG is still a configuration (perhaps, with less concurrency) of the unfolding of the modified STG, i.e., for each instance e of t in Π, Π contains sufficiently many concurrent to e events with labels in U, which can be executed (without firing other instances of t) to supply the missing tokens in the newly inserted place needed to fire e.

Proposition 1 (Liveness): Let $U \overset{n}{\dashrightarrow} t$ be a concurrency reduction transforming a consistent, input-proper and weakly fair STG Υ into Υ', such that t is not a transition of an input signal and for each t-labelled event e and each maximal configuration $\Pi \supseteq [e]$ of the unfolding of Υ there is a finite set $E_U \subseteq \Pi$ of events with labels in U concurrent to e such that $Tokens(\langle e \rangle) + |E_U| > 0$. Then Υ' is a valid implementation of Υ. $\lozenge$

The above liveness condition conveys that no essential behaviour is eliminated by a concurrency reduction. However,

[5] Note that because of this condition, concurrency reduction cannot be performed independently in subsystems, as a deadlock can be introduced. Hence, if a concurrency reduction is performed in some subsystem, it has to be reflected in the STGs for all the other subsystems depending on it.

it employs infinite objects and thus is not directly checkable; hence the tool of [28] had to rely on human input. In this section we propose an approximate test for this condition, which has been implemented in the MPSAT tool. Since we work with safe STGs, MPSAT also implements an additional validity condition stating that the modified STG is safe. Below we separately consider safeness and liveness.

Proposition 2 (Safeness): Let $U \stackrel{n}{\dashrightarrow} t$ be a concurrency reduction transforming a consistent, input-proper, weakly fair and safe STG Υ into Υ', such that t is not a transition of an input signal and the following conditions hold for a complete unfolding prefix of Υ:

(S1) $n = \begin{cases} 0 & \text{if } \#_t[e] = 0 \text{ for some } U\text{-labelled event } e; \\ 1 & \text{otherwise.} \end{cases}$

(S2) $Tokens([e]) = 1$ for each U-labelled event e.

(S3) No two U-labelled events e, f are concurrent.

(S4) $Tokens([e]) = Tokens(C^e)$ for each cut-off event e with a corresponding configuration C^e.

Then Υ' is safe. $\Diamond$

Unfortunately, checking the liveness condition turns out to be much more complicated. In fact, we are not aware of any tool that can do such a check for the full class of safe Petri nets. In particular, PETRIFY simply requires that (i) no events become dead, and (ii) no (new) deadlocks appear [29]. One can easily see that this test is not conservative. Below we propose a more elaborate approximate test (it is also not conservative) based on Proposition 1.

Let $U \stackrel{n}{\dashrightarrow} t$ be a concurrency reduction transforming a consistent, input-proper, weakly fair and safe STG Υ into Υ', such that t is not a transition of an input signal. Then we check the following conditions hold for a finite and complete unfolding prefix of Υ:

(L1) For each t-labelled event e, $Tokens(\langle e \rangle) \geq 0$.

(L2) For each t-labelled event e, if $Tokens(\langle e \rangle) = 0$ then every maximal configuration $C \supseteq [e]$ contains a U-labelled event concurrent to e.

(L3) $Tokens([e]) = Tokens(C^e)$ for each cut-off event e with a corresponding configuration C^e.

The condition (L2) of this test resembles Proposition 1, but a finite and complete prefix is used instead of the full (infinite) unfolding, and a U-labelled event e_U providing a token needed for the t-labelled event to fire is required to be already in the prefix (which is conservative). The only point when this test fails to be conservative is the rare situation when the infinite configuration Π in Proposition 1 is such that truncating it down to events of the prefix results in a non-maximal (w.r.t. $\subset$) configuration, i.e., e_U can be disabled by some event of Π that is not in the prefix. This test, though approximate, seems to work very well in experiments (the author is not aware of any 'practical' STG where it fails, though artificial examples can be constructed).

Computing valid concurrency reductions

One can see that the naïve approach enumerating all the concurrency reductions and filtering them using the safeness and liveness tests described above is not satisfactory because of the combinatorial explosion in the number of possible concurrency reductions (as U can be any non-empty set of transitions). In practice relatively few reductions are valid, and below we describe a method of efficiently computing them using incremental SAT. This method works in two stages. First, concurrency reductions satisfying (S1)–(S4), (L1) and (L3) are computed using incremental SAT. Then the condition (L2) is checked for each of these reductions using another incremental SAT run. We now explain these two stages in more detail.

Stage 1: For each transition t, the valid concurrency reductions of the form $U \stackrel{n}{\dashrightarrow} t$ are computed separately. In useful concurrency reductions, each transition $u \in U$ should be concurrent to t, denoted $u \| t$, i.e., some reachable marking should concurrently enable u and t. We denote by $\mathbf{t}_\|$ the set of transitions concurrent to t (since the STG is consistent, $t \| t$ cannot hold, i.e., $t \notin \mathbf{t}_\|$), and U will be a subset of $\mathbf{t}_\|$. Note that $\mathbf{t}_\|$ can be easily computed using the prefix.

In the SAT instance formulated as the conjunction of constraints given below, we create a variable u tracing whether $u \in U$ for each transition $u \in \mathbf{t}_\|$. Any satisfying assignment A of this SAT instance will correspond to the concurrency reduction for which $U \stackrel{\text{df}}{=} \{u \in \mathbf{t}_\| \mid A(\mathsf{u}) = 1\}$.

- $U \neq \emptyset$:
$$\bigvee_{u \in \mathbf{t}_\|} \mathsf{u} \ .$$
- (S1): $n = \begin{cases} 0 & \text{if } \#_t[e]{=}0 \text{ for some } U\text{-labelled event } e; \\ 1 & \text{otherwise.} \end{cases}$
$$\mathsf{n} \iff \bigwedge_{u \in \mathbf{t}_{\not\parallel}} \neg \mathsf{u} \ ,$$
where $\mathbf{t}_{\not\parallel} \subseteq \mathbf{t}_\|$ is the set of transitions which have instances not preceded by an instance of t, and n is the variable tracing the value of n.
- (S2): $Tokens([e]) = 1$ for each U-labelled event e. We define $Tok_C \stackrel{\text{df}}{=} Tokens(C) \bmod 2$ and $Tok_e \stackrel{\text{df}}{=} Tok_{[e]} \bmod 2$, and provide a modulo-2 formulation of this constraint:
$$\bigwedge_{u \in \mathbf{t}_\|} \bigwedge_{\substack{e \text{ is } u\text{-} \\ \text{labelled}}} (\mathsf{u} \Rightarrow Tok_e) \ .$$
Also, the following defining constraint is added for each Tok_e occurring in the SAT instance:
$$Tok_e \iff \mathsf{n} \oplus (\#_t[e] \bmod 2) \oplus \bigoplus_{\substack{u \in \mathbf{t}_\| \\ \#_u[e] \text{ is odd}}} \mathsf{u} \ .$$
- (S3): no two transitions in U can be concurrent:
$$\bigwedge_{\substack{u, u' \in \mathbf{t}_\| \\ u \| u'}} (\neg \mathsf{u} \vee \neg \mathsf{u}') \ .$$

- (L1): For each t-labelled event e, $Tokens(\langle e \rangle) \geq 0$. We conservatively replace this constraint by $Tokens(\langle e \rangle) = n$ and use a modulo-2 formulation:

$$\bigwedge_{e \in E_t} (\mathsf{n} \iff Tok_{\langle e \rangle}) = \bigwedge_{e \in E_t} (\mathsf{n} \oplus Tok_e) \ ,$$

  where E_t is the set of t-labelled events.
- (S4) = (L3): For each cut-off event e with the corresponding configuration C^e, $Tokens([e]) = Tokens(C^e)$. Again, we use the modulo-2 formulation:

$$\bigwedge_{e/C^e} (Tok_e \iff Tok_{C^e}) \ .$$

The constraints follow the safeness and liveness conditions (except (L2), which is checked separately), and the only potential problem is the use of the modulo-2 formulation for some of the constraints. One can show that such a formulation is nevertheless equivalent to the original one.

To further improve the efficiency of the method, a constraint requiring that a concurrency reduction is potentially useful (i.e., it resolves some conflict core) is also added.

Stage 2: Once all useful concurrency reductions satisfying (S1)–(S4), (L1) and (L3) are computed, the condition (L2) is separately checked for each of them. (L2) holds for $U \stackrel{n}{\dashrightarrow} t$ iff for each instance e of t such that $Tokens(\langle e \rangle) = 0$, the SAT instance described below is unsatisfiable (unfortunately, this condition cannot be incorporated into the SAT instance generated at the first stage).

Intuitively, any satisfying assignment of this instance corresponds to a maximal configuration C of the prefix demonstrating a violation of (L2). This SAT instance has for each event e of the prefix a variable conf_e tracing whether $e \in C$, i.e., for every satisfying assignment A, the set of events $C \stackrel{\text{df}}{=} \{e \mid \mathsf{conf}_e = 1\}$ is a configuration demonstrating the violation of (L2). It is formed as the conjunction of the following constraints:

- C is a configuration of the prefix (not just an arbitrary set of events). Note that C is allowed to contain cut-off events. This condition can be defined as

$$\bigwedge_{e \in E} \bigwedge_{f \in {}^{\bullet\bullet}e} (\mathsf{conf}_e \Rightarrow \mathsf{conf}_f) \wedge \bigwedge_{e \in E} \bigwedge_{f \in ({}^{\bullet}e)^{\bullet} \setminus \{e\}} (\neg \mathsf{conf}_e \vee \neg \mathsf{conf}_f) \ .$$

  The first part of this formula is basically a set of implications ensuring that if $e \in C$ then its immediate predecessors are also in C, i.e., C is downward closed. The second part ensures that C contains no choices.
- C is a maximal configuration of the prefix, i.e., it cannot be extended by any event of the prefix.

$$\bigwedge_{e \in E} \Big(\bigvee_{f \in {}^{\bullet\bullet}e} \neg \mathsf{conf}_f \vee \bigvee_{f \in ({}^{\bullet}e)^{\bullet}} \mathsf{conf}_f \Big) \ .$$

  Intuitively, this constraint conveys that some predecessor of e is not in C or either e or some event that is in the choice relation with e is in C, and so C cannot be extended by e.
- $C \supseteq [e]$, i.e., $e \in C$. This constraint is expressed simply as conf_e.
- C contains no U-labelled events concurrent to e:

$$\bigwedge_{f \in U_{\|}} \neg \mathsf{conf}_f \ ,$$

  where $U_{\|}$ is a set of U-labelled events of the prefix concurrent to e.

One can observe that the first two constraints do not depend on e or t, and so the corresponding parts of the SAT instance do not depend on the concurrency reduction being tested. Furthermore, the remaining constraints are comprised of unit clauses only. These observations turn out to be very useful for implementing this test using incremental SAT. Indeed, this test has to be performed for all the concurrency reductions produced by the first incremental run checking the conditions (S1)–(S4), (L1) and (L3). We can again employ the incremental SAT and use the fact that the individual SAT instances share a large common part (the first two constraints), and differ only by unit clauses (the remaining two constraints). As MINISAT2 treats unit clauses in a special way, allowing to remove them during the incremental SAT without having to regenerate the SAT instance, testing (L2) can be efficiently implemented.

Implementation

The described technique has been implemented in the tool MPSAT. A cost function similar to the one described in Section III for signal insertions was used. On every iteration of the encoding conflict resolution procedure, the best according the cost function transformation (a signal insertion or a concurrency reduction) is chosen and applied to the STG, until all the encoding conflicts are resolved.

The conducted experiments showed that using concurrency reduction in addition to signal insertion can significantly improve the area of the circuit, with a relatively small performance penalty. For example, MPSAT was able to reduce the total area of the small assorted benchmarks in Table II down to 482 literals. However, during these experiments the following phenomenon was observed. It turns out that increasing the weight of the 'lock' relation component of the cost function almost always results in reducing the area of the circuit (this is not the case if only signal insertions are used). This indicates that area optimisation is not a well-posed problem if concurrency reduction is allowed,[6] as the tool tries to reduce the area by sequentialising the circuit. This problem can be alleviated by adding additional constraints (e.g., by

[6]Recall the well-known anecdote about linear programming being used to solve the problem of finding a cheapest ration containing the recommended amounts of all the nutrients, with the computed solution containing several liters of vinegar.

jointly optimising latency and area), and we leave it for future investigation.

VII. Conclusions and future work

This paper proposes a new method for resolution of CSC conflicts based on STG unfoldings. The problem is reformulated in terms of Boolean satisfiability, and a tunable heuristic cost function is used to guide the design space exploration towards good solutions.

The presented case studies demonstrate that the proposed approach explores a large design space and is able to find interesting solutions which could not be found by other methods; moreover, the experimental results show that it is quite fast and results in high quality circuits.

As it was mentioned in the introduction, the proposed approach is intended to be used in conjunction with STG decomposition. This work is completed now, and the results are very encouraging [9].

In future work, we intend to make the liveness test for concurrency reductions conservative and to improve the cost function so that concurrency reductions are treated in a more sensible way. Moreover, some other improvements to the cost function are possible, e.g., based on the ideas described in [7], [24]. Also, compositional synthesis in the presence of concurrency reduction needs further investigation.

Appendix

Proof of Proposition 2 (safeness): Suppose Υ' is not safe. Then there is a configuration C in the (full) unfolding of Υ such that $Tokens(C) > 1$. Due to the completeness of the prefix and (S4), one can assume that C is in the prefix. W.l.o.g., C is minimal w.r.t. $\subset$, and, due to (S1), $C \neq \emptyset$. Hence the set C_{max} of causally maximal events of C is not empty, and all these events have labels in U. However, $|C_{max}| \neq 1$ due to (S2) and $|C_{max}| \not> 1$ due to (S3), a contradiction. ■

Proof that the modulo-2 formulation of the check of (S1)–(S4), (L1) and (L3) is equivalent to the original one:

It is easy to see that if some equality holds, it also holds modulo-2, so the 'interesting' direction of the proof is to show that if a concurrency reduction $U \overset{n}{\dashrightarrow} t$ satisfies the modulo-2 formulation then it also satisfies the original one.

For the sake of contradiction, suppose that $U \overset{n}{\dashrightarrow} t$ satisfies the modulo-2 formulation but not the original one. Hence there must be a causally minimal *bad* event e such that either e is U-labelled and $Tokens([e]) \neq 1$ but $Tok_e = 1$ or e is t-labelled and $Tokens(\langle e \rangle) \neq n$ but $Tok_{\langle e \rangle} = n$. Let $k \stackrel{\text{df}}{=} 0$ in the former case and $k \stackrel{\text{df}}{=} n$ in the latter case. Then in either case $Tokens(\langle e \rangle) \neq k$, but $Tok_{\langle e \rangle} = k$, i.e., $Tokens(\langle e \rangle) \in (k + \mathbb{Z}_{even}) \setminus \{k\}$.

Let $C \stackrel{\text{df}}{=} MinKeep(\langle e \rangle, U \cup \{t\})$, where $MinKeep(C, X)$ is a function which, given a configuration C and a set of transitions X, returns the minimal (w.r.t. $\subset$) configuration $C' \subseteq C$ such that all the X-labelled events of C are in C'. Then $Tokens(C) = Tokens(\langle e \rangle)$, and so $C \neq \emptyset$ as $Tokens(\emptyset) \in \{0, 1\}$ and $\{0, 1\} \cap ((k + \mathbb{Z}_{even}) \setminus \{k\}) = \emptyset$ due to $k \in \{0, 1\}$. Moreover, since the U- and t-labelled events are ordered in C (due to (S3) and non-self-concurrency of t), C is either (i) $[f_u]$ or (ii) $[f_t]$ or (iii) $[f_u] \cup [f_t]$ for some U-labelled event f_u and t-labelled event f_t. In the latter case C can be iteratively replaced by $MinKeep(C \setminus \{f_u, f_t\}, U \cup \{t\})$, until a configuration C' of the form (i) or (ii) is obtained. Note that $Tokens(C) = Tokens(MinKeep(C \setminus \{f_u, f_t\}, U \cup \{t\}))$, in particular $C' \neq \emptyset$.

In case (i), $Tokens([f_u]) \in (k + \mathbb{Z}_{even}) \setminus \{k\}$. Since the modulo-2 formulation is true, $Tok_{f_u} = 1$, and so $Tokens([f_u]) \bmod 2 = 1$. Hence $k = 1$, $Tokens([f_u]) = Tokens(\langle e \rangle) \neq 1$, i.e., f_u is a bad event causally preceding e, contradicting the minimality of e.

In case (ii), $Tokens(\langle f_t \rangle) \in (k + 1 + \mathbb{Z}_{even}) \setminus \{k + 1\}$ and so $Tokens(\langle f_t \rangle) \neq n$, since $n \in \{0, 1\}$ and $((k+1+\mathbb{Z}_{even}) \setminus \{k + 1\}) \cap \{0, 1\} = \emptyset$ due to $k \in \{0, n\}$. Hence f_t is a bad event causally preceding e, contradicting the minimality of e.

Hence the modulo-2 formulation is equivalent to the original one. ■

Acknowledgements

The author would like to thank Josep Carmona and Jordi Cortadella for helpful discussions and benchmarks, and to Mark Schaefer for his feedback concerning the developed MPSat tool.

References

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

[2] "International Technology Roadmap for Semiconductors: Design," 2005, URL: `www.itrs.net/Links/2005ITRS/Design2005.pdf`.

[3] T.-A. Chu, "Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications," Ph.D. dissertation, Lab. for Comp. Sci., MIT, 1987.

[4] L. Rosenblum and A. Yakovlev, "Signal Graphs: from Self-Timed to Timed Ones," in *Proc. Int. Workshop on Timed Petri Nets*. IEEE Comp. Soc. Press, 1985, pp. 199–206.

[5] D. Edwards and A. Bardsley, "Balsa: an Asynchronous Hardware Synthesis Language," *The Comp. Journal*, vol. 45, no. 1, pp. 12–18, 2002.

[6] K. v. Berkel, "Handshake Circuits: an Asynchronous Architecture for VLSI Programming," *Int. Series on Par. Comp.*, vol. 5, 1993.

[7] J. Carmona and J. Cortadella, "Encoding large asynchronous controllers with ILP techniques," *TCAD*, vol. 27, no. 1, pp. 20–33, 2008.

[8] J. Carmona, J.-M. Colom, J. Cortadella, and F. Garcia-Valles, "Synthesis of asynchronous controllers using integer linear programming," *TCAD*, vol. 25, no. 9, pp. 1637–1651, 2006.

[9] V. Khomenko and M. Schaefer, "Combining Decomposition and Unfolding for STG Synthesis," in *Proc. ATPN'07*, ser. LNCS, vol. 4546. Springer-Verlag, 2007, pp. 223–243.

[10] J. Esparza, S. Römer, and W. Vogler, "An Improvement of McMillan's Unfolding Algorithm," *FMSD*, vol. 20, no. 3, pp. 285–310, 2002.

[11] V. Khomenko, "Model Checking Based on Prefixes of Petri Net Unfoldings," Ph.D. dissertation, School of Comp. Sci., Newcastle Univ., 2003.

[12] K. McMillan, "Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits," in *Proc. CAV'92*, ser. LNCS, vol. 663. Springer-Verlag, 1992, pp. 164–174.
[13] A. Semenov, "Verification and Synthesis of Asynchronous Control Circuits Using Petri Net Unfolding," Ph.D. dissertation, School of Comp. Sci., Newcastle Univ., 1997.
[14] V. Khomenko, M. Koutny, and A. Yakovlev, "Detecting State Coding Conflicts in STG Unfoldings Using SAT," *Fund. Inf.*, vol. 62, no. 2, pp. 1–21, 2004.
[15] ——, "Logic Synthesis for Asynchronous Circuits Based on Petri Net Unfoldings and Incremental SAT," *Fund. Inf.*, vol. 70, no. 1–2, pp. 49–73, 2006.
[16] A. Madalinski, A. Bystrov, V. Khomenko, and A. Yakovlev, "Visualization and Resolution of Coding Conflicts in Asynchronous Circuit Design," *IEE Proc.: Comp. & Digital Techniques*, vol. 150, no. 5, pp. 285–293, 2003.
[17] V. Khomenko, "Behaviour-Preserving Transition Insertions in Unfolding Prefixes," in *Proc. ATPN'07*, ser. LNCS, vol. 4546. Springer-Verlag, 2007, pp. 204–222.
[18] ——, "Efficient automatic resolution of encoding conflicts using STG unfoldings," in *Proc. ACSD'07*. IEEE Comp. Soc. Press, 2007, pp. 137–146.
[19] ——, "Efficient automatic resolution of encoding conflicts using STG unfoldings," School of Computing Science, Newcastle University, Tech. Rep. CS-TR-995, 2007, URL: `homepages.cs.ncl.ac.uk/victor.khomenko/papers/papers.html`.
[20] J. Gramm, J. Guo, F. Huffner, and R. Niedermeier, "Data Reduction, Exact, and Heuristic Algorithms for Clique Cover," in *Proc. ALENEX'06*. SIAM, 2006, pp. 86–94.
[21] I. Wegener, *The Complexity of Boolean Functions*. Wiley-Teubner Series in Comp. Sci., 1987.
[22] N. Eén and N. Sörensson, "Translating Pseudo-Boolean Constraints into SAT," *J. on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 1–25, 2006.
[23] J. Savage, "An Algorithm for the Computation of Linear Forms," *SIAM J. on Computing*, vol. 3, pp. 150–158, 1974.
[24] P. Vanbekbergen, F. Catthoor, G. Goossens, and H. De Man, "Optimized Synthesis of Asynchronous Control Circuits from Graph-Theoretic Specifications," in *Proc. ICCAD'90*. IEEE Comp. Soc. Press, 1990, pp. 184–187.
[25] J. Esparza and P. Jančar, "On the Complexity of Consistency and Complete State Coding for Signal Transition Graphs," in *Proc. ACSD'06*. IEEE Comp. Soc. Press, 2006, pp. 47–56.
[26] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proc. of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.
[27] J. Carmona and J. Cortadella, "Private Communication," 2006.
[28] V. Khomenko, A. Madalinski, and A. Yakovlev, "Resolution of encoding conflicts by signal insertion and concurrency reduction based on STG unfoldings," *Fund. Inf.*, vol. 86, no. 3, pp. 299–323, 2008.
[29] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Automatic Handshake Expansion and Reshuffling Using Concurrency Reduction," in *Proc. HWPN'98*, 1998, pp. 86–110.
[30] B. Lin, C. Ykman-Couvreur, and P. Vanbekbergen, "A General State Graph Transformation Framework for Asynchronous Synthesis," in *Proc. EURO-DAC'94*. IEEE Comp. Soc. Press, 1994, pp. 448–453.
[31] J. Carmona, J. Cortadella, and E. Pastor, "A Structural Encoding Technique for the Synthesis of Asynchronous Circuits," *Fund. Inf.*, vol. 50, no. 2, pp. 135–154, 2002.
[32] M. Silva, E. Teruel, and J. Colom, *Lectures on Petri Nets I: Basic Models*, ser. LNCS. Springer-Verlag, 1998, vol. 1491, ch. Linear Algebraic and Linear Programming Techniques for the Analysis of Place/Transition Net Systems, pp. 309–373.

Victor Khomenko obtained an M.Sc. with distinction in Computer Science, Applied Mathematics and Teaching of Mathematics and Computer Science from Kiev Taras Shevchenko University, in 1998, and PhD in Computing Science from Newcastle University, in 2003.

Since September 2005, he has been a Royal Academy of Engineering / EPSRC Post-doctoral Research Fellow, working on the Design and Verification of Asynchronous Circuits (DAVAC) project.

Dr. Khomenko is a Program Committee member for the International Conferences on Application and Theory of Petri Nets and Other Models of Concurrency (ATPN) and International Conference on Application of Concurrency to System Design (ACSD). He also organised the workshops on Unfolding and Partial Order Techniques (UFO'07) and BALSA Re-Synthesis (RESYN'09).