Efficient Automatic Resolution of Encoding Conflicts Using STG Unfoldings

Victor Khomenko*
School of Computing Science, Newcastle University, UK.
Victor.Khomenko@ncl.ac.uk

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 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. 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.*

1. Introduction

Asynchronous circuits are a promising type of digital circuits. They have lower power consumption and electromagnetic emission, no problems with clock skew and related subtle issues, and are fundamentally more tolerant of voltage, temperature and manufacturing process variations. The International Technology Roadmap for Semiconductors report on Design [1] 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 [7, 9] is one of the commonly used tools for synthesis of asynchronous circuits. As a specification it accepts a *Signal Transition Graph (STG)* [6] — 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 may (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 gate library at hand.

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 [10] and TANGRAM [2]. 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 given gate library or exceeding memory or time constraints) are implemented using the syntax-directed translation. The initial experiments conducted in [5] showed that this combined approach can half the area devoted to control flow and improve its latency, compared with the traditional syntax-directed translation, as long as the size of clusters which can be confidently handled by logic syntax is 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 quality of the

*V. Khomenko is a Royal Academy of Engineering/EPSRC Post-Doctoral Research Fellow.

resulting circuit (in terms of area and latency) depends to a large extent on the way the new signals were inserted.

The design flow advocated in [5] 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 [3]. (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, [5] reports that it took 28.3 minutes to resolve CSC conflicts in an STG with 436 places, 398 transitions and 199 signals, followed by 44.7 minutes of synthesis. Moreover, an ILP approximation of the state space may work poorly for some STGs, e.g., those containing many 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 [20] 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 [12, 14, 22].

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.

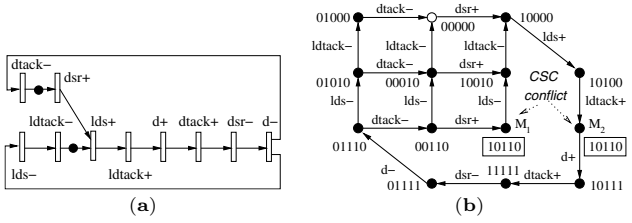
Due to its structural properties (such as acyclicity), the reachable states of an STG can be represented using *configurations* of its unfolding. A configuration C is a downward-closed set of events (being downward-closed means that if $e \in C$ and f is a causal predecessor of e then $f \in C$) without *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 partial-order 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 $[e]$ [24] 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* [12]. Fig. 1(c) shows a finite and complete unfolding prefix of the STG shown in Fig. 1(a); the only cut-off event depicted as a double box, and its corresponding configuration is $\{e_1, e_2\}$.

Efficient algorithms exist for building such prefixes [12, 14], 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 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 [17] they are just slightly bigger than the original STGs themselves. Thus, unfolding prefixes are well-suited for alleviating the state space explosion.

In [17] the unfolding technique was applied to detection of CSC conflicts between reachable states of an STG. Moreover, in [18] the problem of complex-gate logic synthesis from an STG free from CSC conflicts was solved. The experiments in [17, 18] showed that unfolding-based approach can handle much bigger STGs then PETRIFY.

The visualisation method presented in [21] is aimed at facilitating a manual refinement of an STG with CSC con-



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

The order of signals in the binary encodings: *dsr, ldtack, dtack, lds, d*

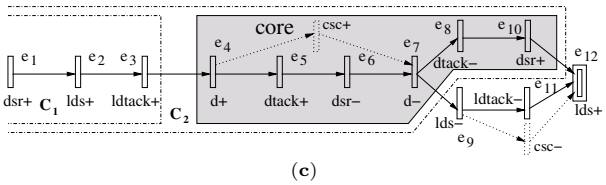


Figure 1. An STG modelling the read cycle of the VME bus controller (a), its state graph showing a CSC conflict between the reachable states M_1 and M_2 (b), and its unfolding prefix showing the conflict core corresponding to this CSC conflict and a way to resolve it by insertion of a new internal signal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

flicts, 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. (A core can be computed as the difference of two configurations whose final states are in CSC conflict.) All such cores must eventually be eliminated 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* [6, 9] — 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 5.)

The semi-automatic approach of [21] is only feasible for synthesis of relatively 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 [15, 17, 18], 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 [5]. Combined with the decomposition approach of [20, 26, 27], this design cycle can be applied for control re-synthesis of BALSA or TANGRAM specifications as described above.

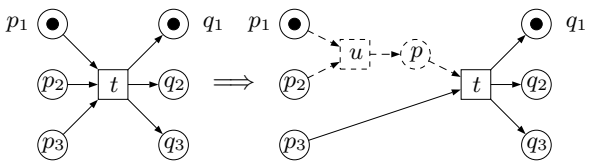
The full version of this paper can be found in the technical report [16] (available on-line), where also the differences of the proposed approach from other existing techniques are discussed.

2. Transformations

In this paper, we are primarily interested in *SB-preserving* transformations, i.e., ones preserving safeness and behaviour (in the sense that the original and the transformed STGs are weakly bisimilar, provided that the newly inserted transitions are considered silent) of the STG. Below we describe several kinds of transition insertions, which we will use for CSC conflict resolution, and the algorithms presented in [15] allow to check their validity.

Building an unfolding prefix of an STG can be a time-consuming operation. However, in most practical cases the approach described in [15] allows to avoid a potentially expensive re-unfolding after each transition insertion, by introducing 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.

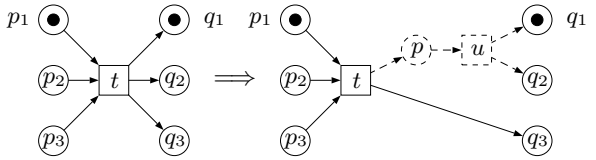
Sequential pre-insertion A sequential pre-insertion is essentially a generalised transition splitting, and is defined as follows. Given a transition t and a set of places $S \subseteq {}^\bullet t$, the sequential pre-insertion $S \wr t$ is the transformation inserting a new transition u (with an additional 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) [15]. Given an unfolding prefix, it is quite easy to check whether a pre-insertion is SB-preserving [15].

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 not be an input transition. Moreover, one should take care that the semi-modularity is not violated. ([15] presents an algorithm allowing one to check that the newly inserted transition will not be in a dynamic choice relation with any other transition, which ensures semi-modularity.)

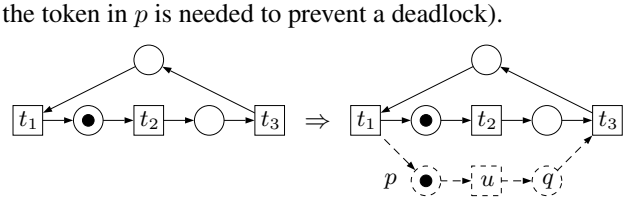
Sequential post-insertion Similarly to sequential pre-insertion, sequential post-insertion is also a generalisation of transition splitting, and is defined as follows. Given a transition t and a set of places $S \subseteq t^\bullet$, the sequential post-insertion $t \wr S$ is the transformation inserting 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.

If a sequential post-insertion is applied to the STG, the semi-modularity is guaranteed to be preserved. 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 $t' \overset{n}{\vdash\!\!\rightarrow} t''$ 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 $t' \vdash\!\!\rightarrow t''$ instead of $t' \overset{0}{\vdash\!\!\rightarrow} t''$ and $t' \overset{\bullet}{\vdash\!\!\rightarrow} t''$ instead of $t' \overset{1}{\vdash\!\!\rightarrow} t''$. The picture below illustrates the concurrent insertion $t_1 \overset{\bullet}{\vdash\!\!\rightarrow} t_3$ (note that the token in p is needed to prevent a deadlock).



In general, concurrent insertions preserve neither safeness nor behaviour. In fact, safeness is not preserved even if $n = 0$ (e.g., when in the original net t' can fire twice without t'' firing), and deadlocks can be introduced even if $n = 1$ (e.g., when in the original net t'' should fire twice before t' can become enabled). In [15], an efficient test whether a concurrent insertion is SB-preserving, working on an unfolding prefix, has been developed.

If a concurrent insertion $t' \overset{n}{\vdash\!\!\rightarrow} t''$ is applied to the STG, the semi-modularity is guaranteed to be preserved, but the inserted transition should not 'delay' an input, and so t'' must not be an 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 other transition insertion;
- a sequential pre-insertion and a sequential post-insertion always commute;
- two sequential pre-insertions $S \wr t$ and $S' \wr t'$ commute iff $t \neq t'$ or $S \cap S' = \emptyset$;
- two sequential post-insertions $t \wr S$ and $t' \wr S'$ commute iff $t \neq t'$ or $S \cap S' = \emptyset$.

It is important to note that an SB-preserving transition insertion remains SB-preserving if another commuting SB-preserving transition insertion is applied first. Hence transformations whose validity has been checked can be cached, and after some transformation has been applied, the non-commuting transformations are removed from the cache and the new transformations that became possible in the modified 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.

3. 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, semi-modularity-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 signal insertions if $\max\bigcup_{t\in T}\{|{}^\bullet t|, |t^\bullet|\}$ is bounded by a constant.) 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 and introduce no auto-conflicts or self-triggering) 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 auto-conflict or one of them can trigger the other.

All these conditions can be checked statically on the prefix, 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 [13]), 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 [11, 28].

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, semi-modularity, 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. 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 [15].) 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:

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

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.

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 $t \wr \cdot$ and $t \stackrel{n}{\mid\!\!\rightarrow} \cdot$ 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:

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

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

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

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 $\mathcal{CS}$ be a core. A signal insertion I is called *hanging w.r.t. $\mathcal{CS}$* if, after it is applied, it directly precedes or succeeds $\mathcal{CS}$. A composite transition insertion $\mathcal{I}$ is *hanging w.r.t. $\mathcal{CS}$* if some $I \in \mathcal{I}$ is hanging w.r.t. $\mathcal{CS}$.

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

$\mathcal{CS}$ 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. $\mathcal{CS}$. By introducing new auxiliary variables $Hanging_{\mathcal{CS}}$ and $Resolved_{\mathcal{CS}}$ for each core $\mathcal{CS}$, the $\mathcal{CORE}$ constraint is defined as follows:

$$\left(\bigvee_{\mathcal{CS}} Resolved_{\mathcal{CS}}\right) \wedge \bigwedge_{\mathcal{CS}} \left(Hanging_{\mathcal{CS}} \iff \bigvee_{I \in H_{\mathcal{CS}}} I\right) \wedge$$

$$\bigwedge_{\mathcal{CS}} \left(\mathit{Resolved}_{\mathcal{CS}} \iff \left(\neg \mathit{Hanging}_{\mathcal{CS}} \wedge \bigoplus_{\substack{I \notin H_{\mathcal{CS}} \wedge \\ \#_I \mathcal{CS} \text{ is odd}}} I \right) \right),$$

where $H_{\mathcal{CS}}$ is the set of hanging w.r.t. $\mathcal{CS}$ 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 [28, 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* [14, Chapter 4] was used. Experiments show that it works quite well in practice.

Another optimisation is to use $x_i \vee x_j$ instead of $x_i \oplus x_j$ for variables which are known to be mutually exclusive (e.g., those corresponding to concurrent or non-commutative transformations).

4. 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 become CSC cores once new signals are added to the STG);
- the estimated delay introduced by the insertion;

[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.

1	$\wr lds^+, \quad \wr dtack^+$
2	$\wr d^-, \quad \wr lds^+$
3	$\wr lds^+, \quad dtack^+ \dashv\!\!\rightarrow d^-$
4	$ldtack^- \wr, \quad \wr dtack^+$
5	$\wr lds^+, \quad \wr dtack^-$
6	$\wr d^-, \quad ldtack^- \wr$
7	$ldtack^- \wr, \quad dtack^+ \dashv\!\!\rightarrow d^-$
8	$\wr d^-, \quad \wr lds^+, \quad \wr dtack^+, \quad \wr dtack^-$
9	$d^+ \dashv\!\!\rightarrow d^-, \quad \wr lds^+$
10	$d^+ \dashv\!\!\rightarrow d^-, \quad ldtack^- \wr$
11	$lds^- \dashv\!\!\rightarrow lds^+, \quad \wr dtack^+$
12	$\wr d^-, \quad lds^- \dashv\!\!\rightarrow lds^+$
13	$\wr lds^+, \quad dsr^- \dashv\!\!\rightarrow dtack^-$
14	$lds^- \dashv\!\!\rightarrow lds^+, \quad dtack^+ \dashv\!\!\rightarrow d^-$
15	$\wr lds^+, \quad dtack^+ \dashv\!\!\rightarrow dtack^-$
16	$d^+ \dashv\!\!\rightarrow d^-, \quad lds^- \dashv\!\!\rightarrow lds^+$
17	$d^+ \dashv\!\!\rightarrow dtack^-, \quad \wr lds^+$

Table 1. The composite transition insertions resolving the CSC conflict shown in Fig. 1.

- 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. The implementation details can be found in the technical report [16].

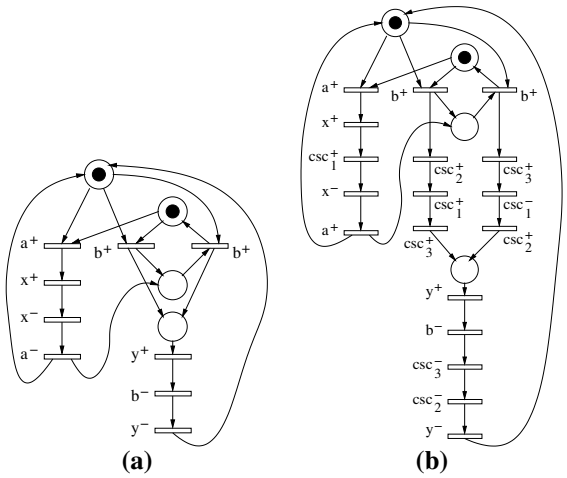
5. Case studies and experimental results

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,

[2]Two signals are in the 'lock' relation [25] 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 logic simplification [4].

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



inputs: a, b; **outputs:** x, y; **internal:** csc_1, csc_2, csc_3

Figure 2. An STG from [5] (a) and a way to resolve the CSC conflicts in it by inserting three signals without restructuring (b).

Example	STG $\|P\|/\|T\|$	*In/Out*	Signals Pfy	ILP	SAT	Literals Pfy	ILP	SAT
ADFAST	15/12	3/3	**2**	**2**	**2**	**14**	17	21
IRCV-BM	55/46	5/4	2	4	**1**	38	46	**28**
MMU	20/16	4/4	**3**	**3**	**3**	29	**27**	**27**
MMU0	20/16	4/4	**3**	5	**3**	29	33	**27**
MMU1	24/16	4/4	3	**2**	**2**	32	**25**	**25**
MR0	31/22	5/6	**3**	4	**3**	45	34	**29**
MR1	25/18	4/5	4	4	**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	**13**	14
PAR(4)	23/20	5/5	**4**	**4**	**4**	**32**	**32**	**32**
SEQ(8)	36/36	9/9	4	6	**3**	47	**43**	44
TSEND-BM	45/39	5/4	2	3	**1**	39	40	**27**
ALLOC-OUTBOUND	17/18	4/3	**2**	**2**	**2**	16	16	**15**
DUPLICATOR	14/12	2/2	**2**	3	**2**	19	16	**13**
MOD4_COUNTER	16/16	1/2	**2**	4	**2**	**25**	28	**25**
RAM-READ-SBUF	26/20	5/5	**1**	**1**	**1**	**18**	19	19
SBUF-RAM-WRITE	29/20	5/5	**2**	**2**	**2**	**22**	29	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**	4	**2**	36	41	**34**
SEQ_MIX	20/20	4/4	3	**2**	**2**	**20**	**20**	**20**
SPEC_SEQ(4)	20/20	5/5	3	3	**2**	**20**	**20**	**20**
Total			51	62	**44**	601	599	**548**

Table 2. Experimental results: assorted small STGs.

MPSAT was able to find 17 possible ways to resolve it, listed in Table 1. This shows that the proposed method explores a fairly large design space, including quite an unintuitive solution 8 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 [5], as the class of transformations it uses is limited to transition splitting.

An 'unresolvable' conflict. The STG in Fig. 2(a) was presented in [5]. PETRIFY can resolve all the CSC conflicts in it by restructuring the net and inserting two signals, and it was claimed that it is impossible to resolve CSC conflicts without such a restructuring. However, MPSAT has found a solution with three signals requiring no restructuring, shown in Fig. 2(b). (When this was reported to the authors of [5], they amended their ILP tool and it was able to resolve the conflicts by inserting four signals.)

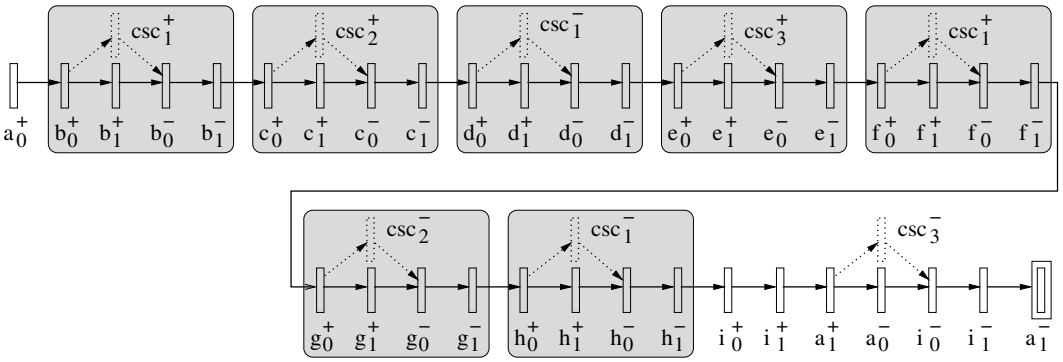
An 8-way sequencer. Sequencers are among the standard 'building blocks' of circuits produced from hardware description languages like BALSA and TANGRAM. The 'parent' handshake at port a initiates eight sequentially ordered 'child' handshakes at ports $b, \ldots, i$. Then the parent handshake completes, 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. 3 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 contains four new signals. MPSAT was able to find a fully concurrent solution with three signals shown in Fig. 3 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 [5] cannot find such a solution because it analyses just a single violation trace on each iteration — in fact, it needs six signals to resolve the CSC conflicts in this case study.

Assorted small benchmarks Table 2 compares the three methods for resolving CSC conflicts: the state-space based approach implemented in PETRIFY, the ILP approach of [5] and the one proposed in this paper, on a number of assorted small benchmarks from [5]. 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 [5] 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 [5], and, for consistency with [5], PETRIFY was used to synthesise the STGs after the CSC conflicts were resolved.

It should be noted that 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, whereas in the latter it was aimed at minimising the number of literals in



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

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

Example	STG $\|P\|/\|T\|$	*In/Out*	Prefix $\|B\|/\|E\|$	Signals Pfy	Signals SAT	Literals Pfy	Literals SAT	Time, [s] Pfy	Time, [s] SAT^s	Time, [s] SAT^l
Marked Graphs										
PpWk(2,3)	23/14	0/7	41/24	1	1	35	34	<1	<1	<1
PpWk(2,6)	47/26	0/13	119/63	1	1	71	70	5	<1	<1
PpWk(2,9)	71/38	0/19	233/120	1	1	107	106	34	<1	6
PpWk(2,12)	95/50	0/25	383/195	1	1	142	142	368	<1	18
PpWk(3,3)	34/20	0/10	63/36	2	2	59	54	4	<1	<1
PpWk(3,6)	70/38	0/19	183/96	2	2	112	108	105	<1	6
PpWk(3,9)	106/56	0/28	357/183	2	2	163	162	1838	4	55
PpWk(3,12)	142/74	0/37	585/297	—	2	—	216	*mem*	5	175
STGs with Arbitration										
PpArb(2,3)	48/32	2/13	110/66	2	2	81	81	35	<1	2
PpArb(2,6)	72/44	2/19	218/120	2	3	117	116	118	1	17
PpArb(2,9)	96/56	2/25	362/192	2	2	153	152	1041	2	50
PpArb(2,12)	120/68	2/31	542/282	—	3	—	188	*mem*	8	159
PpArb(3,3)	71/48	3/19	188/114	3	3	136	131	620	<1	14
PpArb(3,6)	107/66	3/28	368/204	3	3	190	184	5043	2	117
PpArb(3,9)	143/84	3/37	602/321	3	4	244	238	12307	7	354
PpArb(3,12)	179/102	3/46	890/465	—	5	—	292	*mem*	24	839

Table 3. Experimental results: scalable pipelines.

the final implementation.

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 in average 8.5–8.8% smaller implementations. This may seem not particularly spectacular, but such an improvement over Petrify on small benchmarks is noteworthy.

Scalable benchmarks We also compared the described method with Petrify (the ILP tool of [5] 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 [17], with the latter series modified by 'factoring out' the arbiter into the environment to ensure semimodularity. 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).

The results for these two groups are summarised in Table 3, where the meaning of the columns is the same as in Table 2, 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 for each of the methods in the last four columns (for MPSat, 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 tool described in [18], which currently does not support multi-level Boolean minimisation and outputs the equations in the minimised disjunctive normal form (DNF).

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 signi-

ficantly 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.

6. 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 for use in conjunction with the decomposition method of [20, 26, 27]. This work is finished now, and the results are very encouraging [20].

In future work, we intend to extend the method to other transformations, in particular concurrency reduction [8, 19]. Moreover, there is still a scope to improve the cost function, e.g., using the ideas described in [25].

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

References

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

[2] K. v. Berkel. Handshake Circuits: an Asynchronous Architecture for VLSI Programming. *International Series on Parallel Computation*, 5, 1993.

[3] J. Carmona and J. Cortadella. ILP Models for the Synthesis of Asynchronous Control Circuits. In *Proc. DAC'03*, pages 818–826. IEEE Comp. Soc. Press, 2003.

[4] J. Carmona and J. Cortadella. Private Communication, 2006.

[5] J. Carmona and J. Cortadella. State Encoding of Large Asynchronous Controllers. In *Proc. DAC'06*, pages 939–944. IEEE Comp. Soc. Press, 2006.

[6] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, Lab. for Comp. Sci., MIT, 1987.

[7] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. PETRIFY: a Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers. *IEICE Transactions on Information and Systems*, E80-D(3):315–325, 1997.

[8] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Automatic Handshake Expansion and Reshuffling Using Concurrency Reduction. In *Proc. HWPN'98*, pages 86–110, 1998.

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

[10] D. Edwards and A. Bardsley. BALSA: an Asynchronous Hardware Synthesis Language. *The Computer Journal*, 45(1):12–18, 2002.

[11] N. Eén and N. Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–25, 2006.

[12] J. Esparza, S. Römer, and W. Vogler. An Improvement of McMillan's Unfolding Algorithm. *FMSD*, 20(3):285–310, 2002.

[13] J. Gramm, J. Guo, F. Huffner, and R. Niedermeier. Data Reduction, Exact, and Heuristic Algorithms for Clique Cover. In *Proc. ALENEX'06*, pages 86–94. SIAM, 2006.

[14] V. Khomenko. *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD thesis, School of Computing Science, Newcastle University, 2003.

[15] V. Khomenko. Behaviour-Preserving Transition Insertions in Unfolding Prefixes. In *Proc. ATPN'07*, LNCS. Springer-Verlag, 2007. To appear.

[16] V. Khomenko. Efficient Automatic Resolution of Encoding Conflicts Using STG Unfoldings. Technical Report CS-TR-995, School of Computing Science, Newcastle University, 2007. URL: `homepages.cs.ncl.ac.uk/victor.khomenko/papers/papers.html`.

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

[18] 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):49–73, 2006.

[19] V. Khomenko, A. Madalinski, and A. Yakovlev. Resolution of Encoding Conflicts by Signal Insertion and Concurrency Reduction Based on STG Unfoldings. *Fund. Inf.*, 2007. Submitted paper.

[20] V. Khomenko and M. Schaefer. Combining Decomposition and Unfolding for STG Synthesis. In *Proc. ATPN'07*, LNCS. Springer-Verlag, 2007. To appear.

[21] 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):285–293, 2003.

[22] K. McMillan. Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits. In *Proc. CAV'92*, LNCS 663, pages 164–174. Springer-Verlag, 1992.

[23] J. Savage. An Algorithm for the Computation of Linear Forms. *SIAM Journal on Computing*, 3:150–158, 1974.

[24] A. Semenov. *Verification and Synthesis of Asynchronous Control Circuits Using Petri Net Unfolding*. PhD thesis, School of Computing Science, Newcastle University, 1997.

[25] P. Vanbekbergen, F. Catthoor, G. Goossens, and H. De Man. Optimized Synthesis of Asynchronous Control Circuits from Graph-Theoretic Specifications. In *Proc. ICCAD'90*, pages 184–187. IEEE Comp. Soc. Press, 1990.

[26] W. Vogler and B. Kangsah. Improved Decomposition of Signal Transition Graphs. Technical Report 2004-08, Institut für Informatik, Universität Augsburg, 2004.

[27] W. Vogler and R. Wollowski. Decomposition in Asynchronous Circuit Design. Technical Report 2002-05, Institut für Informatik, Universität Augsburg, 2002.

[28] I. Wegener. *The Complexity of Boolean Functions*. Wiley-Teubner Series in Computer Science, 1987.