Derivation of Monotonic Covers for Standard-C Implementation Using STG Unfoldings

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

Abstract

The behaviour of asynchronous circuits is often described by Signal Transition Graphs (STGs), which are Petri nets whose transitions are interpreted as rising and falling edges of signals. One of the crucial problems in the synthesis of such circuits is deriving the set and reset covers for the state-holding elements implementing each output signal of the circuit. The derived covers must satisfy certain correctness constraints, in particular the Monotonic Cover condition *must hold for the standard-C implementation.*

The covers are usually derived using state graphs. In this paper, we avoid constructing the state graph of an STG, which can lead to state space explosion, and instead use a finite and complete prefix of its unfolding. We propose an efficient algorithm for deriving the set and reset covers for the standard-C implementation based on the Incremental Boolean Satisfiability (SAT) approach.

Experimental results show that this technique leads not only to huge memory savings when compared with the methods based on state graphs, but also to significant speedups in many cases, without affecting the quality of the solution.

1. Introduction

Asynchronous circuits are a promising type of digital circuits. They have lower power consumption and electromagnetic emission, no problems with the 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.

Various architectures are used to implement speed-independent circuits [6]. The following three are probably the most well-known (see Fig. 2): (i) the *complex-gate (CG) implementation*, where every output or internal signal in the circuit is implemented as a single (possibly very complicated) atomic gate [4]; (ii) *gC implementation* [13], where each signal is implemented using a pseudo-static latch called *generalised C element (gC element)*; and (iii) *standard-C (stdC) implementation* [2], where each signal is implemented using a C-latch controlled by set and reset signals, which we assume are implemented as complex-gates. The latter two architectures are superficially similar, but one should bear in mind that a gC element is assumed to be atomic, while in the stdC implementation the gates controlling a C-latch have delays. Hence a naïve transformation of a gC implementation into an stdC one can result in a hazardous circuit (see below).

PETRIFY [5, 6] is one of the commonly used tools for synthesis of asynchronous circuits. As a specification it accepts a *Signal Transition Graph (STG)* [4, 17] — 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 circuit with more than 20–30 signals with PETRIFY is often impossible.) Hence, this approach does not scale.

In order to alleviate this problem, Petri net analysis techniques based on causal partial order semantics, in the form of Petri net unfoldings, were applied to circuit synthesis. In [11] the unfolding technique was applied to detection of encoding conflicts between reachable states of an STG, and in [10] a technique for resolution of such conflicts was developed. Moreover, in [12] the problem of complex-gate logic synthesis from an STG free from encoding conflicts was solved. The experiments in [10–12] showed that unfolding-based approach has significant advantage both in memory consumption and in runtime compared with the existing state space based methods; in particular it can handle much bigger STGs than PETRIFY, while delivering circuits with comparable (and often better) area and latency. These techniques essentially complete the design cycle for complex-gate synthesis of asynchronous cir-

*This research was supported by Royal Academy of Engineering / EPSRC grant EP/C53400X/1 (DAVAC).

cuits from STGs that does not involve building reachability graphs at any stage and yet is a fully fledged logic synthesis.

This paper is an extension of the method of [12] to the other mentioned target architectures, viz. gC and stdC implementations. It turns out that gC synthesis is a relatively straightforward generalisation of the approach of [12], and hence this paper focuses mainly on stdC synthesis (gC synthesis is described in the technical report [9]). Although the techniques employed in this paper resemble those in [12], there is a significant and technically difficult new contribution: the stdC synthesis turns out to be much more complicated than CG or gC syntheses since the derived covers must satisfy the *Monotonic Cover condition* [2, 6], which ensures that the resulting circuit is not hazardous. To tackle this problem, the method described in this paper has to derive not only the truth table of the cover, but also a set of *entrance constraints* (which are Boolean implications on the values of the set and reset functions at different points). These constraints, together with the truth table, are then passed to a binate Boolean minimiser to obtain a monotonic cover. Another (minor) contribution of this paper is a better algorithm for computation of the supports (see Section 3.3). The full version of this paper can be found in the technical report [9] (available on-line).

2. Basic definitions

In this section, we present basic definitions concerning Petri nets [16], STGs [5, 6, 17], net unfoldings [7, 14, 18] and Boolean satisfiability [15].

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

A *net system* is a pair $\Sigma \stackrel{\text{df}}{=} (N, M_0)$ comprising a finite net $N = (P, T, F)$ and an (initial) marking M_0. We assume the reader is familiar with the standard notions of the theory of Petri nets (see, e.g., [16]), such as *enabling* and *firing* of a transition, *firing sequence*, marking *reachability*, *deadlock*, and net *boundedness* and *safeness*.

Signal Transition Graphs A *Signal Transition Graph (STG)* is a triple $\Gamma \stackrel{\text{df}}{=} (\Sigma, Z, \lambda)$ such that $\Sigma = (N, M_0)$ is a net system, Z is a finite set of signals, generating the finite alphabet $Z^\pm \stackrel{\text{df}}{=} Z \times \{+, -\}$ of *signal transition labels*, and $\lambda : T \to Z^\pm$ is a labelling function. The signal transition labels are of the form z^+ or z^-, and denote a transition of a signal $z \in Z$ from 0 to 1 (rising edge), or from 1 to 0 (falling edge), respectively. We use the notation $z^\pm$ to denote a transition of signal z if we are not particularly interested in its direction. Γ inherits the operational semantics of its underlying net system Σ, including the notions of transition enabling and firing, reachable markings and firing sequences.

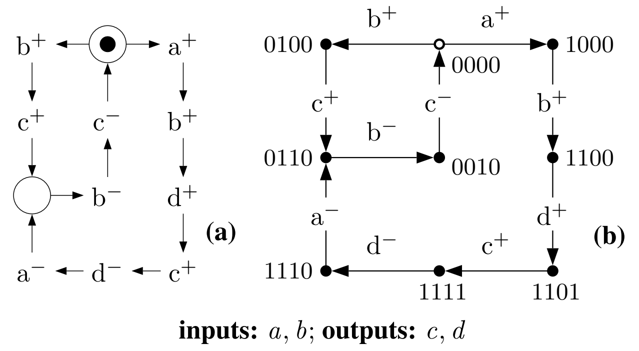

Figure 1. An STG from [2] (a) and its state graph (b). The order of signals in the binary encodings is: a, b, c, d.

We associate with the initial marking of Γ a binary vector $v^0 \stackrel{\text{df}}{=} (v_1^0, \ldots, v_{|Z|}^0) \in \{0, 1\}^{|Z|}$, where each v_i^0 is the initial value of the signal $z_i \in Z$. Moreover, with any finite sequence of transitions σ we associate an integer *signal change vector* $v^\sigma \stackrel{\text{df}}{=} (v_1^\sigma, v_2^\sigma, \ldots, v_{|Z|}^\sigma) \in \mathbb{Z}^{|Z|}$, so that each v_i^σ is the difference between the number of the occurrences of z_i^+–labelled and z_i^-–labelled transitions in σ.

Γ is *consistent* if, for every reachable marking M, all firing sequences σ from M_0 to M have the same *encoding vector* $Code(M)$ equal to $v^0 + v^\sigma$, and this vector is binary, i.e., $Code(M) \in \{0, 1\}^{|Z|}$. We denote by $Code_z(M)$ the component of $Code(M)$ corresponding to a signal $z \in Z$.

The *state graph* SG_Γ of Γ is a tuple $(S, A, M_0, Code)$ such that: S is the set of *states*, which are the reachable markings of Γ; $A \stackrel{\text{df}}{=} \{M \xrightarrow{\lambda(t)} M' \mid M \in S \wedge M \xrightarrow{t} M'\}$ is the set of *state transitions*; M_0 is the *initial state*; and $Code : S \to \{0, 1\}^{|Z|}$ is the *state assignment* function, as defined above for markings.

The signals in Z are partitioned into input signals, Z_I, and output signals, Z_O (the latter may also include internal signals). Input signals are assumed to be generated by the environment, while output signals are produced by the circuit. For each signal $z \in Z_O$, the functions Out_{z^+}, Out_{z^-} and Out_z are defined as follows: $Out_{z^+/z^-/z}(M)$ is 1 if M enables $z^+/z^-/z^\pm$, and 0 otherwise.

In what follows, we assume that the STG is safe, deadlock-free, deterministic (i.e., none of its reachable markings enables two distinct transitions labelled with the same sig-

nal), consistent and semi-modular (i.e., if at some reachable marking a firing of some transition t' disables some other transition t'' then $\lambda(t') \in Z_I$ and $\lambda(t'') \in Z_I$, i.e., a choice is allowed only between inputs). All these properties can be efficiently checked using Petri net unfoldings [18], without building the state graph at any stage.

The Boolean *next-state function* Nxt_z is defined for every reachable state M of Γ and every $z \in Z_O$ as $Nxt_z(M) \stackrel{\text{df}}{=} Code_z(M) \oplus Out_z(M)$, where $\oplus$ is the 'exclusive or' operation. Similarly, the *set* and *reset functions* Set_z and $Reset_z$ are defined as follows:

$$Set_z/Reset_z(M) \stackrel{\text{df}}{=} \begin{cases} 1 & \text{if } Out_{z^+/z^-}(M) = 1 \\ 0 & \text{if } Nxt_z(M) = 0/1 \\ - & \text{otherwise,} \end{cases}$$

where '$-$' denotes the 'don't care' value (i.e., the value of the function can be chosen arbitrarily, with the view of simplifying the resulting implementation). For the circuit to be implementable, the values of Set_z and $Reset_z$ must be uniquely determined by the encoding of each reachable state, i.e., they should be functions of $Code(M)$ rather than M: $Set_z/Reset_z(M) = S_z/R_z(Code(M))$ for some Boolean functions $S_z, R_z : \{0,1\}^Z \to \{0,1\}$.

Note that while any Boolean functions S_z and R_z satisfying the above conditions can be directly used for gC implementation, they must in addition satisfy the *Monotonic Cover condition* [2,6], in order to provide a hazard-free stdC implementation. This condition states that *a cover must be entered only via the states enabling the output z*. To illustrate the importance of this condition, consider the implementation shown in Fig. 2(d), which does not satisfy it, since the state 0110 (which is covered by the set function $\bar{a}b \vee d$ and does not enable c) can be reached from the state 1110 (which is not covered by this set function and does not enable c). Consider the sequence of states $1111 \xrightarrow{d^-} 1110 \xrightarrow{a^-} 0110 \xrightarrow{b^-} 0010$. The gate computing the set function is high at 1111. Firing of d^- drives its output low, but before it reaches 0, a^- can fire, driving its output high; similarly, before it reaches 1, b^- can fire, driving it low. Hence, this gate can exhibit runt non-digital pulses, which may cause the circuit to malfunction.

To address this issue, we introduce the *strict* versions of the set and reset functions as $Set^s_z(M)/Reset^s_z(M) \stackrel{\text{df}}{=} Out_{z^+/z^-}(M)$, and the associated functions S^s_z and R^s_z depending only on $Code(M)$. Note that though these functions are guaranteed to satisfy the Monotonic Cover condition and thus yield a correct stdC implementation of z, it is often not optimal due to the reduced number of 'don't cares' in the truth table, which can be exploited by the Boolean minimiser to simplify the resulting expression. Therefore, the proposed method only uses S^s_z and R^s_z to overapproximate the supports of S_z and R_z satisfying the Monotonic Cover condition, and derives an stdC implementation using a different technique, often yielding better solutions.

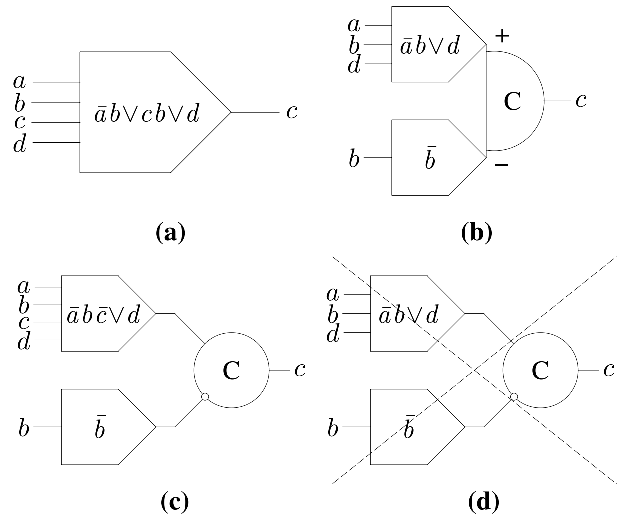

Figure 2. Implementation of signal c of the STG in Fig. 1 using the complex-gate (a), gC (b) and stdC (c) architectures; the result of a naïve transformation of the gC implementation to stdC implementation (d).

Note that in general these functions are incompletely specified because not all the possible encodings occur in the state graph. An incompletely specified Boolean function can be characterised by its ON, OFF and DC ('don't care') sets of inputs on which it evaluates to 1, 0 and '$-$' respectively. These sets must be pairwise disjoint, and their union must contain every possible input.

It can happen that two semantically different states have the same encoding, which means that the circuit is not implementable. To capture this, we define a specific kind of an encoding conflict, called a *strict Complete State Code (sCSC) conflict*. (This notion will be essential for our unfolding-based method of overapproximating the supports of the monotonic covers for stdC implementation.) Suppose $X \subseteq Z$, $z \in Z_O$ and M' and M'' are two distinct states of SG_Γ such that $Code_x(M') = Code_x(M'')$ for all $x \in X$. Then M' and M'' are in $sCSC^{z^+}_X$ / $sCSC^{z^-}_X$ *conflict* if $Out_{z^+/z^-}(M') \neq Out_{z^+/z^-}(M'')$. Γ satisfies the $sCSC^{z^+}$ / $sCSC^{z^-}$ property if no two states of SG_Γ are in $sCSC^{z+}_Z$ / $sCSC^{z^-}_Z$ conflict. Intuitively, this means that the strict set/reset function of z is implementable as a complex gate. It turns out (see the technical report [9] for the proof) that the (not necessarily strict) set and reset functions satisfying the Monotonic Cover condition can be derived (i.e., z is stdC-implementable) iff both $sCSC^{z^+}$ and $sCSC^{z^-}$ properties hold. In fact, the notion of implementability of a signal is invariant in the CG, gC and stdC architectures, i.e., if a signal is implementable in one of them, it is implementable in the other two architectures as well. (In particular, Γ satisfies the traditional CSC property iff it satisfies the $sCSC^{z^+}$ and $sCSC^{z^-}$ properties for each $z \in Z_O$.) CSC conflicts can be detected [11] and resolved [10] on unfold-

$Code(M)$ $abcd$	$S_c(M)$	$R_c(M)$	$S_d(M)$	$R_d(M)$
0100	1	0	0	–
0000	0	–	0	–
1000	0	–	0	–
0110	–	0	0	–
0010	0	1	0	–
1100	0	–	1	0
1110	–	0	0	–
1111	–	0	0	1
1101	1	0	–	0
Expression	$\bar{a}b \vee d$	$\bar{b}$	$ab\bar{c}$	c
Entrance constraints	$S_c(0110) \Rightarrow S_c(1110)$ $S_c(1110) \Rightarrow S_c(1111)$	$\varnothing$	$\varnothing$	$R_d(0110) \Rightarrow R_d(1110)$ $R_d(0110) \Rightarrow R_d(0100)$ $R_d(0010) \Rightarrow R_d(0110)$ $R_d(0100) \Rightarrow R_d(0000)$ $R_d(0000) \Rightarrow R_d(0010)$ $R_d(1000) \Rightarrow R_d(0000)$
Monotonic cover	$\bar{a}b\bar{c} \vee d$	$\bar{b}$	$ab\bar{c}$	cd

Table 1. The truth table for the set and reset functions of output signals of STG in Fig. 1 and the entrance constraints for the stdC implementation.

ings, without building the state graph, and in what follows, we assume that the STG satisfies the CSC property.

One can see that if no two states of Γ are in $sCSC_X^{z^+}$/ $sCSC_X^{z^-}$ conflict then S_z^s/R_z^s can be consistently defined at each state M of SG_Γ as a function of $Code(M)$ restricted to X, i.e., X is a *support* of S_z^s/R_z^s. A support X is *minimal* if no set $Y \subset X$ is a support. (In general, incompletely specified functions can have several distinct minimal supports.) A set $X \subseteq Z$ which is not a support is called a *non-support*. Note that X is a non-support of S_z^s/R_z^s iff the STG has a $sCSC_X^{z^+}/sCSC_X^{z^-}$ conflict.

An example of an STG is shown in Fig. 1(a). It satisfies the CSC property and hence can be implemented using either of the three target architectures considered in this paper. The gC implementation can be obtained by applying Boolean minimisation to the truth table shown in the upper part of Table 1. The first column of this table lists the encodings of all the states of SG_Γ, while the other columns give the corresponding values of the set and reset functions for the output signals. Note that not all possible encodings are present in the first column because the number of reachable states (9) is smaller than the number of possible encodings ($2^4 = 16$). This means that the missing encodings belong to the DC sets of the functions being derived. The next row of the table gives for each output signal of the circuit the result of Boolean minimisation, viz. the set and reset functions which are suitable for the gC implementation. However, as it was explained earlier, these covers do not satisfy the Monotonic Cover condition and thus are not suitable for the stdC implementation. For this circuit the Monotonic Cover condition can be expressed by additional *entrance constraints* shown in the lower part of Table 1, which the produced covers must satisfy; the resulting expressions for the set and reset functions suitable for the stdC implementation are shown in the last row of the table.[1] These functions were obtained as the result of conditional Boolean minimisation, taking into account the entrance constraints; it is reducible to the *binate covering problem* [8].

This essentially completes the synthesis procedure based on state graphs. However, it often leads to state space explosion, and in the proposed approach we follow another way of representing the behaviour of STGs, viz. STG *unfoldings* [7, 14, 18].

STG unfoldings A *finite and complete unfolding prefix* π of an STG Γ is a finite acyclic net which implicitly represents all the reachable states of Γ together with transitions enabled at those states. Intuitively, it can be obtained through *unfolding* Γ, by successive firings of transitions, under the following assumptions: (a) for each new firing a fresh transition (called an *event*) is generated; (b) for each newly produced token a fresh place (called a *condition*) is generated. The full unfolding is infinite whenever Γ has an infinite firing sequence; however, if Γ has finitely many reachable states then the unfolding eventually starts to repeat itself and can be truncated (by identifying a set E_{cut} of *cut-off* events beyond which the prefix is not generated) without loss of information, yielding a finite and complete prefix. We denote by E and B the sets of all events and conditions of π, correspondingly, and by $h : E \cup B \to T \cup P$ the mapping from the nodes of the prefix to the corresponding nodes of the STG. The *size* $|\pi|$ of π is defined as the total number of its events, conditions and arcs.

Due to its structural properties (such as acyclicity), the reachable markings of Γ can be represented using *configurations* of π. A configuration C is a causally closed set of events (being causally closed means that if $e \in C$ and $f \in {}^{\bullet\bullet}e$, then $f \in C$) without *choices* (i.e., for all distinct events $e, f \in C$, ${}^\bullet e \cap {}^\bullet f = \varnothing$). Intuitively, a configuration is a partially ordered execution, i.e., an execution where the order of firing of some of the events (viz. concurrent ones) is not important.

After starting π from the implicit initial marking (with a single token in each condition which does not have an incoming arc) and executing all the events in C, one reaches the marking denoted by $Cut(C)$, and called a *cut*. Then $Mark(C) \overset{\text{df}}{=} h(Cut(C))$ is the marking of Γ which corresponds to this cut. It is remarkable that each reachable marking of Γ is $Mark(C)$ for some configuration C, and, conversely, each configuration C generates a reachable marking $Mark(C)$. This property is a primary reason why various behavioural properties of Γ can be re-stated as the corresponding properties of π, and then checked, of-

[1] Note that in this example the set function for d happens to coincide with next-state function of d. In such a case stdC synthesis allows for a CG implementation of a signal. The precise condition is that if S_z evaluates to 1 in all states M such that $Nxt_z(M) = 1$ (respectively, R_z evaluates to 1 in all states M such that $Nxt_z(M) = 0$) then S_z (respectively, $\overline{R_z}$) can be used as a CG implementation of z.

ten much more efficiently. We extend the functions $Code$, $Code_z$, Nxt_z and $Out_{z/z^+/z^-}$ to configurations of π as follows: $F(C) \stackrel{\text{df}}{=} F(Mark(C))$.

Efficient algorithms exist for building finite and complete prefixes [7], which ensure that the number of non-cut-off events in the resulting prefix never exceeds the number of reachable states of Γ. Moreover, complete prefixes are often exponentially smaller than the corresponding state graphs, especially for highly concurrent Petri nets, because they represent concurrency directly rather than by multidimensional 'diamonds' as it is done in state graphs. For example, if the original Petri net consists of 100 transitions which can fire once in parallel, the state graph will be a 100-dimensional hypercube with 2^{100} vertices, whereas the complete prefix will coincide with the net itself. Since 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; the experimental results in Table 2 demonstrate that high levels of compression can indeed be achieved in practice. Thus, unfolding prefixes are well-suited for alleviating the state space explosion.

Boolean satisfiability The *Boolean satisfiability problem (SAT)* consists in finding a *satisfying assignment*, i.e., a mapping $A : Var \to \{0,1\}$ defined on the set of variables Var occurring in a given Boolean expression φ such that φ evaluates to 1. This expression is often assumed to be given in the *conjunctive normal form (CNF)* $\bigwedge_{i=1}^{n} \bigvee_{l \in L_i} l$, i.e., it is represented as a conjunction of *clauses*, which are disjunctions of *literals*, each literal l being either a variable or the negation of a variable. It is assumed that no two literals within the same clause correspond to the same variable.

Contemporary SAT solvers, e.g., ZCHAFF [15], can be used in the *incremental mode,* i.e., after solving a particular SAT instance the user can slightly modify it (e.g., by adding and/or removing a small number of clauses) and execute the solver again. This is often much more efficient than solving these related instances as independent problems, because on the subsequent runs the solver can use some of the useful information collected so far. In particular, such an approach can be used to compute *projections* of assignments satisfying a given formula.

Let $V \subseteq Var$ be a non-empty set of variables occurring in a formula φ, and $Proj_V^{\varphi}$ be the set of all restricted assignments (or projections) $A|_V$ such that A is a satisfying assignment of φ. Using the incremental SAT approach it is possible to compute $Proj_V^{\varphi}$ as follows.

$\mathcal{PROJ} \leftarrow \varnothing$
while φ is satisfiable **do**
  $A \leftarrow$ a satisfying assignment of φ
  $\mathcal{PROJ} \leftarrow \mathcal{PROJ} \cup \{A|_V\}$
  Append to φ the clause $\bigvee_{\substack{v \in V \\ A(v)=1}} \neg v \vee \bigvee_{\substack{v \in V \\ A(v)=0}} v$

Suppose now that we are interested in finding only the maximal elements of $Proj_V^{\varphi}$, assuming that $A|_V \le A'|_V$ if $(A|_V)(v) \le (A'|_V)(v)$, for all $v \in V$. The above procedure can then be modified by changing the appended clause to $\bigvee_{\substack{v \in V \\ A(v)=0}} v$; moreover, before terminating, an additional pass over the elements of $\mathcal{PROJ}$ is made in order to eliminate any non-maximal projections.

3. Logic synthesis based on unfoldings

Although the process of logic synthesis described in Section 2 is relatively straightforward, it suffers from the state space explosion problem due to the necessity of constructing the entire state graph of the STG. This section describes a synthesis procedure based on a finite and complete STG unfolding prefix, completely avoiding generation of the state graph. We assume a given consistent STG satisfying the CSC property, and consider in turn each output signal $z \in Z_O$. For the stdC synthesis one has to derive the set and reset functions S_z and R_z satisfying the Monotonic Cover condition. In what follows, we only describe how to derive the set function, since the derivation of the reset function is very similar.

3.1. Outline of the proposed method

The starting point of the proposed approach is to consider the set $\mathcal{NSUPP}$ of all non-supports of S_z^s. (Recall that a set $X \subseteq Z$ is a non-support of S_z^s iff the STG has a $sCSC_X^{z^+}$ conflict.) Within the Boolean formula $\mathcal{CSC}$ which we are going to construct, non-supports are represented by variables $\mathsf{nsupp} \stackrel{\text{df}}{=} \{\mathsf{nsupp}_x \mid x \in Z\}$. The key property of $\mathcal{CSC}$ is that if one fixes the values of the variables nsupp then the resulting formula is satisfiable iff there is a $sCSC_X^{z^+}$ conflict, where $X \stackrel{\text{df}}{=} \{x \mid \mathsf{nsupp}_x = 1\}$. That is, if for a given satisfying assignment A of $\mathcal{CSC}$ the set of signals $\{x \mid A(\mathsf{nsupp}_x) = 1\}$ is identified with the projection $A|_{\mathsf{nsupp}}$ (note that there are other variables besides nsupp in $\mathcal{CSC}$) then $\mathcal{NSUPP} = Proj_{\mathsf{nsupp}}^{\mathcal{CSC}}$. Hence one can use the incremental SAT approach described in Section 2 to compute $\mathcal{NSUPP}$. In fact, it is sufficient for the proposed approach to compute the set of maximal non-supports $\mathcal{NSUPP}_{\max} \stackrel{\text{df}}{=} \max_{\subset} \mathcal{NSUPP}$, which can then be used for computing the set $\mathcal{SUPP}_{\min} \stackrel{\text{df}}{=} \min_{\subset}\{X \subseteq Z \mid X \not\subseteq X', \text{ for all } X' \in \mathcal{NSUPP}_{\max}\}$ of all the minimal supports of S_z^s.

$\mathcal{SUPP}_{\min}$ captures the set of all possible supports of S_z^s, in the sense that any support is an extension of some minimal support, and vice versa, any extension of any minimal support is a support. However, the simplest equation is usually obtained for some minimal support, and this approach was adopted. Yet, this is not a limitation of the proposed method, as one can also explore some or all of the non-minimal supports, which can be advantageous, e.g., for small circuits and/or when the synthesis time is not of paramount importance (this would sometimes allow one to

find a simpler equation). And vice versa, not all minimal supports have to be explored: if some minimal support has many more signals compared with another one, the corresponding expression is likely to be more complicated, and so too large supports can be safely discarded. Thus, as usual, there is a trade-off between the execution time and the degree of design space exploration, and the proposed method allows one to reach an acceptable compromise. Typically, several 'most promising' supports are selected and used for subsequent derivation of the cover, and the simplest cover is chosen in the end.

Suppose now that X is one of the chosen supports of S_z^s. Note that instead of deriving the function S_z^s, we derive the function S_z subject to the entrance constraints. This often results in a better implementation, since the corresponding truth table has more 'don't-cares'. However, the support X used in this case was computed for the function S_z^s, and hence can contain more signals than necessary. This does not result in an inferior implementation, since the Boolean minimisation will remove the redundant signals if this helps to simplify the resulting expression. Note also that the binate covering problem in this case is guaranteed to have a solution, since S_z^s is always a possible solution (however, better solutions can be found by Boolean minimisation).

In order to compute the truth table for S_z and a chosen support X, we build a Boolean formula which has a variable code'_x for each signal $x \in X$ and is satisfiable iff these variables can be assigned values in such a way that there is a configuration C' such that $Code_x(C') = \mathsf{code}'_x$, for all $x \in X$, and $S_z(Code(C')) = 1$ (for the ON set) or 0 (for the OFF set). Then, using the incremental SAT approach, one can compute the corresponding projections of the sets of reachable encodings onto X, which gives the truth table for S_z. In addition to the truth table, a set of entrance constraints of the form $S_z(v) \Rightarrow S_z(v')$ is generated. The computed truth table, together with this set of implications, is then fed to a Boolean minimiser, which completes the synthesis. The minimisation is performed under the supplied constraints, and the problem is known as the *binate covering problem* [8].

It should be noted that the size of the truth table for Boolean minimisation and the number of times a SAT solver is executed in the proposed method can be exponential in the size of the support. Thus, it is crucial for the performance of the proposed algorithm that the support of each function is relatively small. However, in practice it is anyway difficult to implement as an atomic logic gate a Boolean expression depending on more than, say, eight variables. (Atomic behaviour of logic gates is essential for the speed-independence of the circuit, and a violation of this requirement can lead to hazards [4, 6].) This means that if some function has only 'large' supports then the specification must be changed (e.g., by adding new internal signals) to introduce 'smaller' supports. Such transformations are related to the *logic decomposition and technology mapping* step in the design cycle for asynchronous circuits [6]; we do not consider it in this paper.[2]

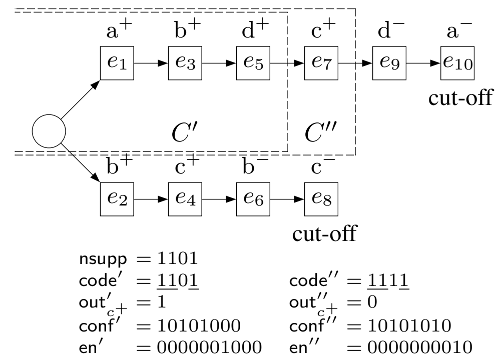

Figure 3. An unfolding prefix of the STG in Fig. 1 illustrating a $sCSC^{c^+}_{\{a,b,d\}}$ conflict between two configurations. The order of signals in the binary encodings is: a, b, c, d.

3.2. Computing maximal non-supports

At the level of a branching process, a $sCSC^{z^+}_X$ conflict can be represented as an unordered *conflict pair* of configurations $\langle C', C'' \rangle$ whose final states are in $sCSC^{z^+}_X$ conflict. Fig. 3 illustrates a $sCSC^{c^+}_{\{a,b,d\}}$ conflict, which means that $\{a, b, d\}$ is a non-support of S_z^s. Note that the set function of the stdC implementation must have an additional signal c in its support, as shown in Fig. 2(c,d), whereas a gC implementation with the support $\{a, b, d\}$ is possible, see Fig. 2(b).

We adopt the following naming conventions for the $\mathcal{CSC}$ formula. The variable names are in the lower case and names of formulae are in the upper case. Names with a single prime (e.g., conf'_e and $\mathcal{CONF}'$) are related to C', and ones with a double prime (e.g., conf''_e) are related to C''. If there is no prime then the name is related to both C' and C''. If a formula name has a single prime then the formula does not contain occurrences of variables with double primes, and the counterpart double prime formula can be obtained from it by adding another prime to every variable with a single prime. The subscript of a variable points to which element of the STG or the prefix the variable is related, e.g., conf'_e and conf''_e are both related to the event e of the prefix. By a variable without a subscript we denote the list of all variables for all possible values of the subscript, e.g., conf' denotes the list of variables conf'_e, where e runs over the set $E \setminus E_{cut}$.

Below we describe the Boolean variables which are used in the proposed translation. Some of them can be expressed via others, and in such a case an appropriate *defining ex-*

[2]This problem is yet to be solved using unfoldings; however, it is an independent problem, and significant progress in this direction has already been made.

pression is provided, and it is assumed that whenever such a variable is used in some formula, the corresponding defining expression is also added to this formula. (And if it, in turn, depends on some other variables with defining expressions, they are also added, and so on.)

- For each event $e \in E \setminus E_{cut}$, we create two Boolean variables, conf'_e and conf''_e, tracing whether $e \in C'$ and $e \in C''$, respectively.
- For each signal $x \in Z$, we create a variable nsupp_x indicating whether x belongs to a non-support.
- For each condition $b \in B \setminus E^{\bullet}_{cut}$, we create two Boolean variables, cut'_b and cut''_b, tracing whether $b \in Cut(C')$ and $b \in Cut(C'')$ respectively. The defining expression for cut'_b is $\mathsf{cut}'_b \iff \bigwedge_{e \in {}^\bullet b} \mathsf{conf}'_e \wedge \bigwedge_{e \in b^\bullet \setminus E_{cut}} \neg \mathsf{conf}'_e$, which conveys that $b \in Cut(C')$ iff the event 'producing' b has fired, but no event 'consuming' b has fired. (Note that since $|{}^\bullet b| \leq 1$, $\bigwedge_{e \in {}^\bullet b} \mathsf{conf}'_e$ in this formula is either the constant 1 or a single variable.) The defining expression for cut''_b can be built similarly.
- For each signal $x \in Z$, we create two Boolean variables, code'_x and code''_x, tracing the values of $Code_x(C')$ and $Code_x(C'')$ respectively.

The following well-known folklore construction allows one to incorporate the current encoding of each signal into the current marking. For each signal $z \in Z$, a pair of complementary places, p^0_z and p^1_z, tracing the value of z is added to the STG. Each z^+–labelled transition has p^0_z in its preset and p^1_z in its postset, and each z^-–labelled transition has p^1_z in its preset and p^0_z in its postset. Exactly one of these two places is marked at the initial state, accordingly to the initial value of signal z (these initial values can be computed using the unfolding).

One can show that at any reachable state of an STG augmented with such places, p^0_z (respectively, p^1_z) is marked iff the value of z is 0 (respectively, 1). Thus, if a transition labelled by z^+ (respectively, z^-) is enabled then the value of z is 0 (respectively, 1), which in turn guarantees the consistency. Such a transformation can be done completely automatically. For a consistent STG, it does not restrict the behaviour and yields an STG with an isomorphic state graph. In what follows, we assume such tracing places in the STG.

We observe that $Code'_x(C') = 1$ iff $p^1_x \in Mark(C')$, i.e., iff $b \in Cut(C')$ for some p^1_x–labelled condition b (note that the places in P_Z cannot contain more than one token). This is captured by the defining expression $\mathsf{code}'_x \iff \bigvee_{b \in B_x} \mathsf{cut}'_b$, where $B_x \stackrel{\text{df}}{=} \{B \setminus E^{\bullet}_{cut} \mid h(b) = p^1_x\}$. (Note that $p^1_x \in Mark(C')$ iff $\bigvee_{b \in B_x} \mathsf{cut}'_b$ is true.) The defining expression for code''_x can be built similarly.

- For each event $e \in E$, we create two Boolean variables, en'_e and en''_e, tracing whether e is 'enabled' by C' and C'' respectively. Note that unlike conf' and conf'', such variables are also created for the cut-off events. The defining expression for en'_e is $\mathsf{en}'_e \iff \bigwedge_{b \in {}^\bullet e} \mathsf{cut}'_b$. Intuitively, it states that e is 'enabled' by C' iff all the conditions in ${}^\bullet e$ are in $Cut(C')$. The defining expression for en''_e can be built similarly.
- For each signal $x \in Z$, we create two Boolean variables, $\mathsf{out}'_{x/x^+/x^-}$ and $\mathsf{out}''_{x/x^+/x^-}$, tracing the values of $Out_{x/x^+/x^-}(C')$ and $Out_{x/x^+/x^-}(C'')$ respectively. The defining expression for $\mathsf{out}'_{x/x^+/x^-}$ is defined as $\mathsf{out}'_{x/x^+/x^-} \iff \bigvee_{e \in E : \lambda(h(e)) = x^\pm/x^+/x^-} \mathsf{en}'_e$. Intuitively, it conveys that $x^\pm/x^+/x^-$ is 'enabled' by C' iff some $x^\pm/x^+/x^-$-labelled event is enabled by C'. The defining expression for $\mathsf{out}''_{x/x^+/x^-}$ can be built similarly.

As already mentioned, the aim is to build a Boolean formula $\mathcal{CSC}$ such that $Proj^{\mathcal{CSC}}_{\mathsf{nsupp}} = \mathcal{NSUPP}$, i.e., after assigning arbitrary values to the variables nsupp, the resulting formula is satisfiable iff there is a $sCSC^{z^+}_X$ conflict, where $X \stackrel{\text{df}}{=} \{x \mid \mathsf{nsupp}_x = 1\}$. Fig. 3 shows a satisfying assignment (except the variables cut' and cut'') corresponding to the $sCSC^{c^+}_{\{a,b,d\}}$ conflict depicted there. The target formula $\mathcal{CSC}$ will be the conjunction of constraints described below.

Configuration constraints The role of first two constraints, $\mathcal{CONF}'$ and $\mathcal{CONF}''$, is to ensure that C' and C'' are legal configurations (not just arbitrary sets of events). $\mathcal{CONF}'$ is defined as the conjunction of two formulae: $\bigwedge_{e \in E \setminus E_{cut}} \bigwedge_{f \in {}^{\bullet\bullet} e} (\mathsf{conf}'_e \Rightarrow \mathsf{conf}'_f)$ (ensuring that C' is a causally closed set of events) and $\bigwedge_{e \in E \setminus E_{cut}} \bigwedge_{f \in (({}^\bullet e)^\bullet \setminus \{e\}) \setminus E_{cut}} \neg(\mathsf{conf}'_e \wedge \mathsf{conf}'_f)$ (ensuring that C' contains no choices). $\mathcal{CONF}''$ is defined similarly.

One can see that the size of the configuration constraints is $O(|E \setminus E_{cut}|^2)$, but since STGs in practice usually contain just a few choices, this upper bound is rather pessimistic. Moreover, it is possible to reduce it down to $O(|\pi|)$, at the expense of introducing auxiliary variables. This linear in the size of the prefix translation is not considered in this paper as it is quite complicated, even though it was implemented in the actual tool. Note that the configuration constraint does not depend on the output signal z being synthesised, and thus it can be re-used many times during the synthesis.

Encoding constraint The role of the encoding constraint $\mathcal{NSUPP}$ is to ensure that $Code_x(C') = Code_x(C'')$ whenever $\mathsf{nsupp}_x = 1$. This is conveyed by the formula $\bigwedge_{x \in Z} \left(\mathsf{nsupp}_x \Rightarrow (\mathsf{code}'_x \iff \mathsf{code}''_x)\right)$, with the appropriate defining expressions.

One can show that under the plausible assumption that for each signal $x \in Z$, x^+- and x^--labelled events occur in the prefix, the size of the encoding constraint is $O(|\pi|)$. Note that the encoding constraint does not depend on the output signal z being synthesised, and thus it can be re-used many times during the synthesis.

Separation constraint The role of the separation constraint $\mathcal{SEP}$ is to ensure that $Out_{z^+}(C') \neq Out_{z^+}(C')$. Since all the other constraints are symmetric w.r.t. C' and C'', one can rewrite it as $Out_{z^+}(C') = 1 \wedge Out_{z^+}(C') = 0$,

which is conveyed by the formula $\mathsf{out}'_{z^+} \wedge \neg\mathsf{out}''_{z^+}$, with the appropriate defining expressions.

Unlike the configuration and encoding constraints, the separation constraint does depend on the output signal z being synthesised. However, under the plausible assumption that for each signal $z \in Z_O$, z^+- and z^--labelled events occur in the prefix, one can ensure (by re-using parts of formulae) that the total size of the generated formulae for all $z \in Z_O$ is $O(|\pi|)$.

Translation to SAT Now the problem of computing the set $\mathcal{NSUPP}_{\max}$ of maximal non-supports of S^s_z can be formulated as a problem of finding the maximal elements of the projection $Proj^{\mathcal{CSC}}_{\mathsf{nsupp}}$ for the Boolean formula $\mathcal{CSC} \stackrel{\text{df}}{=} \mathcal{CONF}' \wedge \mathcal{CONF}'' \wedge \mathcal{NSUPP} \wedge \mathcal{SEP}$. This can be done using the incremental SAT approach described in Section 2. The size of this formula is $O(|\pi|)$.

3.3. Computing minimal supports

In [12] the set $\mathcal{SUPP}_{\min}$ of minimal supports was computed using the incremental SAT approach. Below a better approach is described.

The characteristic function of the set of *all* non-supports can be built as $\bigvee_{X \in \mathcal{NSUPP}_{\max}} \Big(\bigwedge_{x \in Z \setminus X} \neg\mathsf{nsupp}_x \Big)$. One can see that this expression is in the *disjunctive normal form (DNF)*, i.e., it is represented as a disjunction of *monoms*, which are conjunction of *literals*, each literal being either a variable or the negation of a variable. Moreover, this function is *negative unate* in all its variables, i.e., every its literal is the negation of a variable. Since the minimal DNF of a unate function is uniquely defined as the disjunction of all its prime implicants, and the sets in $\mathcal{NSUPP}_{\max}$ are pairwise incomparable w.r.t. $\subset$, the expression above is this minimal DNF.

The characteristic function of the set $\mathcal{SUPP}$ of all supports can be obtained as the negation of the above expression; since the original function is negative unate in all its variables, the result will be *positive unate* in all its variables, i.e., in its minimal DNF no literal is the negation of a variable. Each prime implicant of the characteristic function of $\mathcal{SUPP}$ defines a minimal support, and so one can compute the set $\mathcal{SUPP}_{\min}$ of the minimal supports by building its (uniquely defined) minimal DNF and considering the set of its monoms.

For example, the function S^s_c of the STG shown in Fig. 1 has four maximal non-supports: $\{a,b,c\}$, $\{a,b,d\}$, $\{a,c,d\}$ and $\{b,c,d\}$. The characteristic function of its set of non-supports is $\bar{d} \vee \bar{c} \vee \bar{b} \vee \bar{a}$, and the minimal DNF of the negation of this expression is $abcd$. Hence S^s_c has a single minimal support $\{a,b,c,d\}$.

Note that in general the DNF of the negation of a given DNF expression can be exponentially larger than the original expression, even in the unate case. Nevertheless, in practice it is often much easier to complement a unate function. In the actual implementation we used the `unate_compl` function provided by ESPRESSO [3].

3.4. Computing the truth table

Suppose that X is a (not necessarily minimal) support of S^s_z. For stdC synthesis, one has to compute the truth table (i.e., the ON, OFF and DC sets) for S_z (not S^s_z!) and the support X, and derive the entrance constraints. Even though S_z is being derived, the (potentially larger) support computed for S^s_z is used. This, on the one hand, guarantees that it is possible to derive S_z satisfying the entrance constraints (i.e., the binate covering problem will always have a solution), and, on the other hand, does not result in an inferior implementation since Boolean minimisation will remove the 'redundant' signals from the resulting expression when this leads to simplification.

Note that S_z is not completely specified even on the reachable states of the STG. To reduce the number of incremental calls to the SAT solver, we do not compute the DC set explicitly (it is implicitly defined by the ON and OFF sets), and compute the ON and OFF sets separately as follows.

The ON set of S_z can be computed as the projection of the formula defined as the conjunction of $\mathcal{CONF}'$, the defining expressions for the variables code'_x, $x \in X$, and out'_{z^+} (with the appropriate defining expressions) onto the set of variables $\{\mathsf{code}'_x \mid x \in X\}$. Intuitively, this projection will contain the encodings of all reachable states enabling z^+, restricted to X. Similarly, the OFF set of S_z is computed as the projection of the formula defined as the conjunction of $\mathcal{CONF}'$, the defining expressions for the variables code'_x, $x \in X$, and $\mathsf{code}'_z \iff \mathsf{out}'_z$ (the latter conveys that $Nxt_z(C') = 0$; note that $Nxt_z(C) \equiv \neg Code_z(C) \iff Out_z(C)$), with the appropriate defining expressions, onto the set of variables $\{\mathsf{code}'_x \mid x \in X\}$. Intuitively, this projection will contain the encodings of all reachable states at which Nxt_z is 0, restricted to X.

3.5. Derivation of the entrance constraints

The Boolean minimisation procedure must now take into account the Monotonic Cover condition, i.e., ensure that the entrance constraints are satisfied by the solution. These are a set of implications which can be computed as follows. We construct a formula defined as the conjunction of $\mathcal{CONF}'$, the defining expressions for the variables code'_x, $x \in X$, and $\mathsf{code}'_z \wedge \neg\mathsf{out}'_{z^-} \wedge \bigvee_{x \in X \setminus \{z\}} \mathsf{out}'_x$, with the appropriate defining expressions. One can show that under the plausible assumption that for each signal $x \in Z$, x^+- and x^--labelled events occur in the prefix, the size of this formula is $O(|\pi|)$.

Each satisfying assignment A of this formula defines a configuration C' such that $Code_z(C')$=1, $Out_{z^-}(C')$=0 and enabling some event e labelled by some signal $x \in X \setminus \{z\}$. Let v be the projection of A onto the set of variables $\{\mathsf{code}'_x \mid x \in X\}$. The entrance constraint for the

final state of C' conveys that if the final state of $C' \cup \{e\}$ is in the cover than the final state of C' is also in the cover. That is, for each signal $x \in X \setminus \{z\}$ enabled by C', the implication $S_z(v') \Rightarrow S_z(v)$ should be added to the set of entrance constraints, where v' is obtained from v by negating the bit corresponding to x. Then the clause

$$\bigvee_{\substack{x \in X \setminus \{z\} \\ v_x = 0}} \mathsf{code}'_x \vee \bigvee_{\substack{x \in X \setminus \{z\} \\ v_x = 1}} \neg \mathsf{code}'_x \vee \bigvee_{\substack{x \in X \setminus \{z\} \\ Out_x(C) = 0}} \mathsf{out}'_x,$$

which is not satisfied by A, is appended to the formula (note that all the necessary defining expressions are already in the formula), and the process is repeated until the instance becomes unsatisfiable. Intuitively, this clause eliminates all the satisfying assignments with the corresponding configuration having the same projection v of the encoding of its final state onto the set of variables $\{\mathsf{code}'_x \mid x \in X\}$ and enabling the same or smaller set of signals from $X \setminus \{z\}$.

Eventually we end up with a set of implications of the form $S_z(v') \Rightarrow S_z(v)$. It can happen that v or v' is in the OFF set of the function, and the following simplifications are possible. If v' is in the OFF set then the implication can be deleted, as it is trivially satisfied. If v is in the OFF set then the OFF set is extended to include v' as well (this, in turn, can trigger further simplifications), and the implication can be deleted. The computed ON and OFF sets together with the entrance constraints are fed to the binate Boolean minimisation algorithm [8], which in this case is guaranteed to have solutions.

3.6. Optimisations

In the full version [9] of this paper, we describe optimisations which can significantly reduce the computation effort required by the proposed method. First, we suggest a heuristic allowing to compute a part of a signal's support without running a SAT solver, based on the fact that any support for an output z must include all the *triggers* of z, i.e., those signals whose firing can enable z. Then we show how to speed up the computation in the case of prefixes without choices.

4. Experimental results and conclusions

The proposed method was implemented using ZCHAFF SAT solver [15] and ESPRESSO Boolean minimiser [3], and the benchmarks from [12] were attempted. All the experiments were conducted on a PC with a $Pentium^{TM}$ IV/2.8GHz processor and 512M RAM.

The experimental results are summarised in Table 2, where the meaning of the columns is as follows (from left to right): the name of the problem; the number of places, transitions, and input and output signals in the STG; the number of reachable states; the number of conditions, events and cut-off events in the complete prefix; the total number of next-state (for gC synthesis) or set and reset (for CG and stdC syntheses) functions obtained by the proposed method (it gives a rough idea of the explored design space); the time spent by the PETRIFY tool [5] for each of the three types of synthesis; and the time spent by the method proposed in this paper for each of the three types of synthesis. We use 'mem' if there was a memory overflow and 'time' to indicate that the test had not stopped after 15 hours. The time needed to build complete prefixes is not included in the table, since it did not exceed 0.1sec for any of the attempted STGs.

Note that in all cases the size of the complete prefix was relatively small. This can be explained by the fact that STGs usually contain a lot of concurrency but relatively few choices, and thus the prefixes are in many cases not much bigger then the STGs themselves. For the scalable benchmarks, one can observe that the complete prefixes exhibited polynomial growth, whereas the number of reachable states grew exponentially. As a result, the proposed method had a clear advantage over that based on state graphs. In all the test cases it solved the problem relatively easily, even when it was intractable for PETRIFY. In some cases, it was faster by several orders of magnitude. The time spent on all these benchmarks was quite satisfactory — the 'hardest' benchmark was CFASYMCSCA for all the three types of synthesis, and it took just 56/69/171 seconds for the CG/gC/stdC synthesis algorithm. Such 'hard' cases typically had many alternative implementations which were enumerated by the proposed method, and the rates at which the equations were derived were quite high — e.g., for CFASYMCSCA benchmark in average approximately 8/7/3 equations per second were derived by the CG/gC/stdC synthesis algorithm.

It is important to note that these improvements in memory and running time come without any reduction in quality of the solutions. In fact, the proposed method is *complete*, i.e., it can produce all the valid implementations in each of the three target architectures (CG, gC and stdC). However, in the developed tool we restricted the algorithm to only minimal supports. Nevertheless, the explored design space was quite satisfactory: as the 'Derived expressions' column in Table 2 shows, in many cases the method proposed quite a few alternative implementations. Overall, the proposed approach turned out to be clearly superior, especially for hard problem instances.

In most cases the gC synthesis took more time than CG synthesis, because more formulae are generated and more equations are usually produced; however, the difference in time is not very significant. The stdC synthesis was 1.5–3 times more expensive than gC synthesis, because the entrance constraints had to be built. However, it did not matter much for timing that the binate covering problem had to be solved, as in all cases it was very fast compared with the other stages of the method.

According to the experimental results, the new method can solve quite large problem instances relatively quickly. It should also be emphasised that the unfolding approach

Problem	STG				Prefix			Derived expressions (SAT)			PFY Time, [s]			SAT Time, [s]		
	$\|P\|$	$\|T\|$	$\|Z_I\|/\|Z_O\|$	$\|S\|$	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	CG	gC	stdC	CG	gC	stdC	CG	gC	stdC
Real-Life STGs																
LAZYRING	42	37	5/7	187	88	71	5	14	9/12	9/12	1	3	3	<1	<1	<1
RING	185	172	11/18	16320	650	484	55	63	69/39	69/39	850	849	898	3	5	8
DUP4PHCSC	135	123	12/15	171	146	123	11	48	85/45	85/45	20	27	38	<1	<1	1
DUP4PHMTRCSC	114	105	10/16	149	122	105	8	46	75/29	75/29	13	19	34	<1	<1	1
DUPMTRMODCSC	152	115	10/17	321	228	149	13	165	85/33	85/33	125	141	148	1	1	2
CFSYMCSCA	85	60	8/14	6672	1341	720	56	60	48/80	48/80	163	183	253	22	25	82
CFSYMCSCB	55	32	8/8	690	160	71	6	34	20/12	20/12	10	12	20	<1	<1	<1
CFSYMCSCC	59	36	8/10	2416	286	137	10	18	14/16	14/16	13	15	22	<1	<1	<1
CFSYMCSCD	45	28	4/10	414	120	54	6	16	14/10	14/10	3	7	5	<1	<1	<1
CFASYMCSCA	128	112	8/26	147684	1808	1234	62	450	252/259	252/259	1448	1565	1569	56	69	171
CFASYMCSCB	128	112	8/24	147684	1816	1238	62	93	78/65	78/65	2323	2481	2508	19	24	42
Marked Graphs																
PPWKCSC(2,3)	24	14	0/7	$2^7 = 128$	38	20	1	7	7/7	7/7	<1	2	2	<1	<1	<1
PPWKCSC(2,6)	48	26	0/13	$2^{13} = 8192$	110	56	1	13	13/13	13/13	4	6	6	<1	<1	<1
PPWKCSC(2,9)	72	38	0/19	$2^{19} > 5 \cdot 10^5$	218	110	1	19	19/19	19/19	44	44	44	<1	<1	<1
PPWKCSC(2,12)	96	50	0/25	$2^{25} > 3 \cdot 10^7$	362	182	1	25	25/25	25/25	2082	2055	2056	<1	<1	1
PPWKCSC(3,3)	36	20	0/10	$2^{10} = 1024$	57	29	1	10	10/10	10/10	1	3	3	<1	<1	<1
PPWKCSC(3,6)	72	38	0/19	$2^{19} > 5 \cdot 10^5$	165	83	1	19	19/19	19/19	43	46	46	<1	<1	<1
PPWKCSC(3,9)	108	56	0/28	$2^{28} > 2 \cdot 10^8$	327	164	1	28	28/28	28/28	7380	7085	7080	<1	<1	1
PPWKCSC(3,12)	144	74	0/37	$2^{37} > 10^{11}$	543	272	1	37	37/37	37/37	*time*	*time*	*time*	1	1	2
STGs with Arbitration																
PPARBCSC(2,3)	48	32	2/13	$207 \cdot 2^4 = 3312$	110	66	2	18	13/18	13/18	4	6	6	<1	<1	<1
PPARBCSC(2,6)	72	44	2/19	$207 \cdot 2^{10} > 2 \cdot 10^5$	218	120	2	24	19/24	19/24	42	43	44	<1	<1	<1
PPARBCSC(2,9)	96	56	2/25	$207 \cdot 2^{16} > 10^7$	362	192	2	30	25/30	25/30	315	316	317	<1	<1	1
PPARBCSC(2,12)	120	68	2/31	$207 \cdot 2^{22} > 8 \cdot 10^8$	542	282	2	36	31/36	31/36	3840	3959	3976	1	1	2
PPARBCSC(3,3)	71	48	3/19	$297 \cdot 2^8 = 76032$	118	114	3	29	19/29	19/29	45	47	47	<1	<1	<1
PPARBCSC(3,6)	107	66	3/28	$297 \cdot 2^{17} > 3 \cdot 10^7$	368	204	3	38	28/38	28/38	1001	1176	1175	<1	<1	1
PPARBCSC(3,9)	143	84	3/37	$297 \cdot 2^{26} > 10^{10}$	602	321	3	47	37/47	37/47	24941	25544	25753	1	2	3
PPARBCSC(3,12)	179	102	3/46	$297 \cdot 2^{35} > 10^{13}$	890	465	3	56	46/56	46/56	*mem*	*mem*	*mem*	2	3	5

Table 2. Experimental results.

is particularly well-suited for STGs, because STG unfolding prefixes are much smaller than state graphs for practical STGs. Therefore, in contrast to state-space based approaches, the proposed method is not memory demanding.

We view these results as encouraging. Together with those of [10, 11, 18] they form complete design flows for CG, gC and stdC syntheses of asynchronous circuits based on STG unfolding prefixes rather than state graphs. In future work we intend to include also the logic decomposition and technology mapping step into this framework.

References

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

[2] P. Beerel, C. Myers, and T.-Y. Meng. Covering Conditions and Algorithms for the Synthesis of Speed-Independent Circuits. *IEEE Trans. on CAD*, 1998.

[3] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni Vincentelli. *Logic Minimisation Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

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

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

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

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

[8] A. Grasselli and F. Luccio. Some Covering Problems in Switching Theory. In *Network and Switching Theory*, pages 536–557, 1968.

[9] V. Khomenko. Derivation of Set and Reset Covers for gC Elements and Standard C Implementation Using STG Unfoldings. Technical Report CS-TR-930, School of Comp. Sci., Newcastle Univ., 2005. URL: `http://homepages.cs.ncl.ac.uk/victor.khomenko/papers/papers.html`.

[10] V. Khomenko. Efficient Automatic Resolution of Encoding Conflicts Using STG Unfoldings. In *Proc. ACSD*, pages 147–156, 2007.

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

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

[13] A. Martin. Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. In *Developments in Concurrency and Communication*, UT Year of Prog. Series, pages 1–64, 1990.

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

[15] S. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. CHAFF: Engineering an Efficient SAT Solver. In *Proc. DAC*, pages 530–535, 2001.

[16] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proc. of the IEEE*, 77(4):541–580, 1989.

[17] L. Rosenblum and A. Yakovlev. Signal Graphs: from Self-Timed to Timed Ones. In *Proc. Int. Workshop on Timed Petri Nets*, pages 199–207, 1985.

[18] A. Semenov. *Verification and Synthesis of Asynchronous Control Circuits Using Petri Net Unfolding*. PhD thesis, School of Comp. Sci., Newcastle Univ., 1997.