# Logic Synthesis for Asynchronous Circuits Based on Petri Net Unfoldings and Incremental SAT

Victor Khomenko,[1] Maciej Koutny[1] and Alex Yakovlev[2]
[1]School of Computing Science
[2]School of Electrical, Electronic and Computer Engineering
University of Newcastle upon Tyne, NE1 7RU, United Kingdom
{Victor.Khomenko, Maciej.Koutny, Alex.Yakovlev}@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 equations for logic gates implementing each output signal of the circuit. This is usually done using reachability graphs.*

*In this paper, we avoid constructing the reachability graph of an STG, which can lead to state space explosion, and instead use only the information about causality and structural conflicts between the events involved in a finite and complete prefix of its unfolding. We propose an efficient algorithm for logic synthesis 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 reachability graphs, but also to significant speedups in many cases, without affecting the quality of the solution.*

***Keywords:*** *logic synthesis, asynchronous circuits, self-timed circuits, Petri nets, signal transition graphs, STG, SAT, net unfoldings, partial order techniques.*

## 1. Introduction

Signal Transition Graphs (STGs) is a formalism widely used for describing the behaviour of asynchronous control circuits. Typically, they are used as a specification language for the synthesis of such circuits [2, 3, 14]. STGs are a class of interpreted Petri nets, in which transitions are labelled with the names of rising and falling edges of circuit signals. Circuit synthesis based on STGs involves: (i) checking the necessary and sufficient conditions for the STG's implementability as a logic circuit; (ii) modifying, if necessary, the initial STG to make it implementable; and (iii) finding appropriate Boolean covers for the next-state functions of output and internal signals and obtaining them in the form of Boolean equations for the logic gates of the circuit. The latter step is traditionally called *logic synthesis for complex gates.* One of the commonly used STG-based synthesis tools, PETRIFY [3], performs all of these steps automatically, after first constructing the reachability graph (in the form of a BDD) of the initial STG specification. Since popularity of this tool is steadily growing, it is likely that STGs and Petri nets will increasingly be seen as an intermediate (back-end) notation for the design of large controllers.

While the state-based approach is relatively simple and well-studied, the issue of computational complexity for highly concurrent STGs is quite serious due to the state space explosion problem. This puts practical bounds on the size of control circuits that can be synthesized using such techniques, which are often restrictive, especially if the STG models are not constructed manually by a designer but rather generated automatically from high-level hardware descriptions.

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 [7], we proposed a solution for one of the subproblems, central to the implementability analysis in step (i), viz. checking the Complete State Coding (CSC) condition. In essence, this problem consists in detecting the state encoding conflicts, which occur when semantically different reachable states have the same binary encoding. We showed that the notion of an encoding conflict can be characterized in terms of satisfiability of a Boolean formula (SAT). The algorithms achieved significant speedups compared with methods based on reachability graphs, and provided a basis for the framework for resolution of encoding conflicts (step (ii) above) described in [10], which used the set of pairs of configurations representing encoding conflicts produced by the algorithm as an input.

However, those techniques would have limited practical impact if it was necessary to construct the reachability graph in the later stages of the design cycle for asynchronous circuits. To address this concern, we show in this paper how Petri net unfolding techniques can be used for deriving equations for logic gates of the circuit (step (iii) above). This essentially completes the design cycle for complex gates synthesis that does not involve building reachability graphs at any stage yet is a fully fledged logic synthesis. Our experiments have shown that the proposed method has significant advantage both in memory consumption and in execution time compared with the existing state space based methods, without affecting the quality of the solutions. This paper is a short version of the technical report [8] (available on-line).

## 2. Basic definitions

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. Also, 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., Figure 1(a)). If this hidden place contained a token, it is drawn directly on the arc. For every $t \in T$, we assume that ${}^\bullet t \neq \varnothing \neq t^\bullet$, where ${}^\bullet t \stackrel{\text{df}}{=} \{s \mid (s,t) \in F\}$ and $t^\bullet \stackrel{\text{df}}{=} \{s \mid (t,s) \in F\}$.

A *net system* is a pair $\Sigma \stackrel{\text{df}}{=} (N, M_0)$ comprising a finite net $N = (P,T,F)$ and an (initial) marking $M_0$. A transition $t \in T$ is *enabled* at a marking $M$, denoted $M[t\rangle$, if for every $s \in {}^\bullet t$, $M(s) \geq 1$. Such a transition can be *fired*, leading to a marking $M'$ calculated as the multiset $M' \stackrel{\text{df}}{=} M - {}^\bullet t + t^\bullet$. We denote this by $M[t\rangle M'$ or $M[\rangle M'$ if the identity of the transition is irrelevant. The set of *reachable* markings of $\Sigma$ is the smallest (w.r.t. $\subseteq$) set $[M_0\rangle$ containing $M_0$ and such that if $M \in [M_0\rangle$ and $M[\rangle M'$ then $M' \in [M_0\rangle$. For a finite sequence of transitions $\sigma = t_1 \ldots t_k$, we denote $M[\sigma\rangle M'$ if there are markings $M_0, \ldots, M_k$ such that $M_0 = M$, $M_k = M'$ and $M_{i-1}[t_i\rangle M_i$, for $i = 1, \ldots, k$.

A *Signal Transition Graph (STG)* is a triple $\Gamma \stackrel{\text{df}}{=} (\Sigma, Z, \lambda)$ such that $\Sigma = (N, M_0)$ is a net system, $Z$ is a finite set of signals, generating a finite alphabet $Z^\pm \stackrel{\text{df}}{=} Z \times \{+,-\}$ of *signal transition labels*, and $\lambda : T \to Z^\pm$ is a labelling function. The signal transition labels are of the form $z^+$ or $z^-$, and denote a transition of a signal $z \in Z$ from 0 to 1 (rising edge), or from 1 to 0 (falling edge), respectively. Signal transitions are associated with the actions which change the value of a particular signal. $\Gamma$ inherits the operational semantics of its underlying net system $\Sigma$, including the notions of transition enabling and firing, reachable markings, and firing sequences.

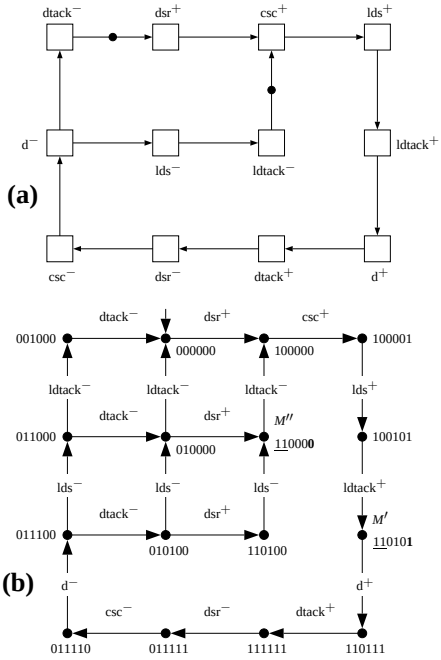



**Figure 1. An STG specifying a VME bus controller (a) with inputs *dsr, ldtack* and outputs *dtack, lds, d, csc*; and its state graph (b) illustrating a $CSC^{csc}_{\{dsr,ldtack\}}$ conflict between the states $M'$ and $M''$ (the order of signals in the binary encodings is: *dsr, ldtack, dtack, lds, d, csc*).**

We associate with the initial marking of $\Gamma$ a binary vector $v^0 \stackrel{\text{df}}{=} (v^0_1, \ldots, v^0_{|Z|}) \in \{0,1\}^{|Z|}$, where each $v^0_i$ corresponds to the signal $z_i \in Z$. Moreover, with any finite sequence of transitions $\sigma$ we associate an integer *signal change vector* $v^\sigma \stackrel{\text{df}}{=} (v^\sigma_1, v^\sigma_2, \ldots, v^\sigma_{|Z|}) \in \mathbb{Z}^{|Z|}$, so that each $v^\sigma_i$ is the difference between the number of the occurrences of $z_i^+$–labelled and $z_i^-$–labelled transitions in $\sigma$.

$\Gamma$ is *consistent* if, for every reachable marking $M$, all firing sequences $\sigma$ 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 consistency can be enforced syntactically, by adding to the STG, for each signal $z \in Z$, a pair of complementary places, $p^0_z$ and $p^1_z$, tracing the value of $z$ as follows. For

each $z^+$–labelled transition $t$, $p_z^0 \in {}^\bullet t$ and $p_z^1 \in t^\bullet$, and for each $z^-$–labelled transition $t'$, $p_z^1 \in {}^\bullet t'$ and $p_z^0 \in t'^\bullet$. Exactly one of these two places is marked at the initial state, accordingly to the initial value of signal $z$. One can show that at any reachable state of an STG augmented with such places, $p_z^0$ (respectively, $p_z^1$) 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 of the augmented STG. 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 (see below); for a non-consistent STG, the transformation restricts the behaviour and may lead to (new) deadlocks. In what follows, we assume that the tracing places are present in the STG, and denote $P_Z^0 \stackrel{\text{df}}{=} \{p_z^0 \mid z \in Z\}$, $P_Z^1 \stackrel{\text{df}}{=} \{p_z^1 \mid z \in Z\}$, and $P_Z \stackrel{\text{df}}{=} P_Z^0 \cup P_Z^1$.

The *state graph* of a consistent STG $\Gamma$ is a tuple $SG_\Gamma \stackrel{\text{df}}{=} (S, A, M_0, Code)$ such that: $S \stackrel{\text{df}}{=} [M_0\rangle$ is the set of *states*, $A \stackrel{\text{df}}{=} \{M \stackrel{t}{\rightarrow} M' \mid M \in [M_0\rangle \wedge M[t\rangle M'\}$ is the set of *state transitions*, $M_0$ is the *initial state*, and $Code : S \rightarrow \{0,1\}^{|Z|}$ 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 logic gates of the circuit. For each signal $z \in Z_O$ we define

$$Out_z(M) \stackrel{\text{df}}{=} \begin{cases} 1 & \text{if } M[t\rangle \text{ and } \lambda(t) \in \{z^+, z^-\} \\ 0 & \text{otherwise.} \end{cases}$$

Logic synthesis derives for each output signal $z \in Z_O$ a Boolean *next-state function* $Nxt_z$ defined for every reachable state $M$ of a consistent STG $\Gamma$ as

$$Nxt_z(M) \stackrel{\text{df}}{=} Code_z(M) \oplus Out_z(M)\ ,$$

where $\oplus$ is the 'exclusive or' operation. The value of this function must be determined without ambiguity by the encoding of each reachable state, i.e., $Nxt_z(M) = F_z(Code(M))$ for some function $F_z : \{0,1\}^Z \rightarrow \{0,1\}$ ($F_z$ will eventually be implemented as a logic gate). To capture this, let $M'$ and $M''$ be two distinct states of $SG_\Gamma$, $z \in Z_O$ and $X \subseteq Z$. $M'$ and $M''$ are in *Complete State Coding conflict for $z$ w.r.t. $X$ ($CSC_X^z$ conflict)* if $Code_x(M') = Code_x(M'')$ for all $x \in X$ and $Nxt_z(M') \neq Nxt_z(M'')$. $\Gamma$ satisfies the *CSC property for $z$ ($CSC^z$ property)* if no two states of $SG_\Gamma$ are in $CSC_Z^z$ conflict, and $\Gamma$ satisfies *the CSC property* if it satisfies the $CSC^z$ property for each $z \in Z_O$. $X$ is a *support* of $z \in Z_O$ if no two states of $\Gamma$ are in $CSC_X^z$ conflict. In such a case the value of $Nxt_z$ at each state $M$ of $SG_\Gamma$ is determined without ambiguity by the encoding of $M$ restricted to $X$. A support $X$ of $z \in Z_O$ is *minimal* if no set $X' \subset X$ is a support of $z$. In general, a signal can have several distinct minimal supports.

An example of a consistent STG for a data read operation in a simple VME bus controller (a standard STG benchmark, see, e.g., [3]) is shown in Figure 1(a). Figure 1(b) shows its state graph, which allows one to derive the equations implementing each output signal by applying Boolean minimization to the truth table shown in Table 1. The first column of this table lists the encodings of all the states of $SG_\Gamma$, while the other columns give the corresponding values of the next-state functions for all the output signals. Note that not all possible encodings are present in the first column, because the number of reachable states (16) is smaller than the number of possible encodings ($2^6 = 64$). This means that the missing encodings form the 'don't care' set, i.e., the values of the functions at these encodings are not important and can be chosen arbitrarily. (Boolean minimization procedures can exploit this to reduce the complexity of the resulting Boolean expression.) The result of Boolean minimization, viz. the expressions computed by logic gates implementing the output signals of the circuit, are as follows: $Nxt_{dtack} = d$, $Nxt_{lds} = d \vee csc$, $Nxt_d = csc \wedge ldtack$, and $Nxt_{csc} = dsr \wedge (\neg ldtack \vee csc)$.

| *Code* | $Nxt_{dtack}$ | $Nxt_{lds}$ | $Nxt_d$ | $Nxt_{csc}$ |
|---|---|---|---|---|
| 001000 | 0 | 0 | 0 | 0 |
| 000000 | 0 | 0 | 0 | 0 |
| 100000 | 0 | 0 | 0 | 1 |
| 100001 | 0 | 1 | 0 | 1 |
| 011000 | 0 | 0 | 0 | 0 |
| 010000 | 0 | 0 | 0 | 0 |
| 110000 | 0 | 0 | 0 | 0 |
| 100101 | 0 | 1 | 0 | 1 |
| 011100 | 0 | 0 | 0 | 0 |
| 010100 | 0 | 0 | 0 | 0 |
| 110100 | 0 | 0 | 0 | 0 |
| 110101 | 0 | 1 | 1 | 1 |
| 011110 | 1 | 1 | 0 | 0 |
| 011111 | 1 | 1 | 1 | 0 |
| 111111 | 1 | 1 | 1 | 1 |
| 110111 | 1 | 1 | 1 | 1 |

**Table 1. The truth table for the output signals of the VME bus controller example (the order of signals in the binary encodings is: *dsr, ldtack, dtack, lds, d, csc*).**

This essentially completes the standard complex-gate synthesis procedure based on state graphs. However, it often leads to state space explosion, and in our approach we follow another way of representing the behaviour of STGs, viz. STG *unfoldings* [4, 5, 11].

A *finite and complete prefix* of an STG's unfolding is a finite acyclic net $\pi$ which implicitly represents all the reachable states of $\Gamma$ together with transitions enabled at those states. The set of places of $\pi$ (called *conditions*) is denoted by $B$, and the set of transitions (called *events*) by $E$; each

condition or event $r$ has its label, $h(r)$, which is a place or transition of $\Gamma$, respectively. The partial order relation on events induced by the (acyclic) flow relation of $\pi$ is denoted by $\preceq$.

Algorithmically, $\pi$ can be obtained through *unfolding* $\Gamma$, by successive firings of transition, under the following rules: (a) for each new firing a fresh event is generated; (b) for each newly produced token a fresh condition is generated. The unfolding is infinite whenever $\Gamma$ has an infinite run; however, if $\Gamma$ has finitely many reachable states then the unfolding eventually starts to repeat itself and can be truncated without loss of information, yielding a finite and complete prefix $\pi$. This truncation is effected by the set of *cut-off* events $E_{cut} \subseteq E$ beyond which no new events are generated. Figure 2 shows a finite and complete prefix for the example STG depicted in Figure 1(a).

Due to its structural properties (such as acyclicity), the reachable markings of $\Gamma$ can be represented using *configurations* of $\pi$. A configuration $C$ is a downward-closed set of events (being downward-closed means that if $e \in C$ and $f \preceq e$, then $f \in C$) without structural conflicts (i.e., for all distinct events $e, f \in C$, ${}^\bullet e \cap {}^\bullet f = \varnothing$). Intuitively, a configuration is a partial-order execution, i.e., an execution where the order of firing of some of the events is not important.

After starting $\pi$ from the implicit initial marking (whereby one puts 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 $Cut(C)$. Then $Mark(C) \stackrel{\text{df}}{=} h(Cut(C))$ is the marking of $\Gamma$ which corresponds to the marking of $\pi$ reached after firing the events in $C$. It is remarkable that each reachable marking of $\Gamma$ 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 $\Gamma$ can be re-stated as the corresponding properties of $\pi$, and then checked, often much more efficiently (in particular, one can easily check the consistency of $\Gamma$ [13]).

One can show that the number of events in the complete prefix can never exceed the number of reachable states of $\Sigma$ [5]. Moreover, complete prefixes are often exponentially smaller than the corresponding reachability graphs, especially for highly concurrent Petri nets, because they represent concurrency directly rather than by multidimensional 'diamonds' as it is done in reachability graphs. For example, if the original Petri net consists of 100 transitions which can fire once in parallel, the reachability graph will contain $2^{100}$ nodes, whereas the complete prefix will coincide with the net itself. The experimental results in Table 2 demonstrate that high levels of compression can indeed be achieved in practice.

Below we extend the functions $Code$, $Code_z$, $Nxt_z$ and $Out_z$ to configurations of $\pi$, as follows: $Code(C) \stackrel{\text{df}}{=} Code(Mark(C))$, $Code_z(C) \stackrel{\text{df}}{=} Code_z(Mark(C))$, $Nxt_z(C) \stackrel{\text{df}}{=} Nxt_z(Mark(C))$, and $Out_z(C) \stackrel{\text{df}}{=} Out_z(Mark(C))$.

## 3. Boolean satisfiability

The *Boolean satisfiability (SAT) problem* 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 $\varphi$ such that $\varphi$ 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 in the same clause correspond to the same variable.

Some of the leading SAT solvers, e.g., ZCHAFF [12], can be used in the *incremental mode*, i.e., after solving a particular SAT instance the user can slightly change 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 (e.g., learnt clauses, see [16]) collected so far. In particular, such an approach can be used to compute *projections* of assignments satisfying a given formula, as described in sequel.

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

**Step 0:** $\mathcal{A} := \varnothing$.

**Step 1:** Run the SAT solver for $\varphi$.

**Step 2:** If $\varphi$ is unsatisfiable then return $\mathcal{A}$ and terminate.

**Step 3:** Add $A|_V$ to $\mathcal{A}$, where $A$ is the satisfying assignment found in Step 1.

**Step 4:** Modify $\varphi$ by appending a new clause $\bigvee_{v \in V \wedge A(v)=1} \neg v \vee \bigvee_{v \in V \wedge A(v)=0} v$.

**Step 5:** Go back to Step 1.

Suppose now that we are interested in finding only the minimal elements of $Proj_V^\varphi$, assuming that $A|_V \leq A'|_V$ if $(A|_V)(v) \leq (A'|_V)(v)$, for all $v \in V$. The above procedure can then be modified by changing Step 4 to:

**Step 4':** Modify $\varphi$ by appending a new clause $\bigvee_{v \in V \wedge A(v)=1} \neg v$.

Similarly, if we were interested in finding all the maximal elements of $Proj_V^\varphi$, then one could change Step 4 to:

**Step 4":** Modify $\varphi$ by appending a new clause $\bigvee_{v \in V \wedge A(v)=0} v$.

Moreover, in the latter two cases, before terminating any non-minimal (or non-maximal) projections should be eliminated from $\mathcal{A}$.

## 4. Logic synthesis based on unfolding prefixes

Although the process of logic synthesis described in Section 2 is straightforward, it suffers from the state space explosion problem due to the necessity of constructing the entire state graph of the STG. In this section, we describe an approach based on unfolding prefixes rather than state graphs. It has been noted in [7] that in practice such prefixes are often much smaller than the corresponding state spaces. This can be explained by the fact that practical 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.

### 4.1. Outline of the proposed method

In [7], the CSC conflict detection problem was solved by reducing it to SAT. More precisely, given a finite and complete prefix of an STG's unfolding, one can build a formula $\mathcal{CSC}$ which is satisfiable iff there is a CSC conflict. In this paper, we modify that construction in the way described below. We assume a given consistent STG satisfying the CSC property, and consider in turn each output signal $z \in Z_O$.

The starting point of the proposed approach is to consider the set $\mathcal{NSUPP}^z$ of all sets of signals which are *non-supports* of $z$. Within the Boolean formula $\mathcal{CSC}^z$, which we are going to construct, non-supports are represented by variables $\mathsf{nsupp} \stackrel{\text{df}}{=} \{\mathsf{nsupp}_x \mid x \in Z\}$, and, for a given assignment $A$, the set of signals $X = \{x \mid A(\mathsf{nsupp}_x) = 1\}$ is identified with the projection $A|_{\mathsf{nsupp}}$ (note that there are other variables besides $\mathsf{nsupp}$ in $\mathcal{CSC}^z$). The key property of $\mathcal{CSC}^z$ is that $\mathcal{NSUPP}^z = Proj^{\mathcal{CSC}^z}_{\mathsf{nsupp}}$, and so it is possible to use the incremental SAT approach to compute $\mathcal{NSUPP}^z$. However, for our purposes it is enough to compute the maximal non-supports $\mathcal{NSUPP}^z_{\max} \stackrel{\text{df}}{=} \max_{\subseteq} \mathcal{NSUPP}^z$ which can then be used for computing the set

$$\mathcal{SUPP}^z_{\min} \stackrel{\text{df}}{=} \min_{\subseteq}\{X \subseteq Z \mid X \not\subseteq X', \text{ for all } X' \in \mathcal{NSUPP}^z_{\max}\}$$

of all the minimal supports of $z$ (another incremental SAT run will be needed for this).

$\mathcal{SUPP}^z_{\min}$ captures the set of all possible supports of $z$, 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 in our experiments. Yet, this is not a limitation of our 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 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 equation will almost certainly be more complicated, and so too large supports can safely be discarded. Thus, as usual, there is a trade-off between the execution time and the degree of design space exploration, and our method allows one to choose an acceptable compromise. Typically, several 'most promising' supports are selected, the equations expressing $Nxt_z$ as a function of signals in these supports are obtained (as described below), and the simplest among them is implemented as a logic gate.

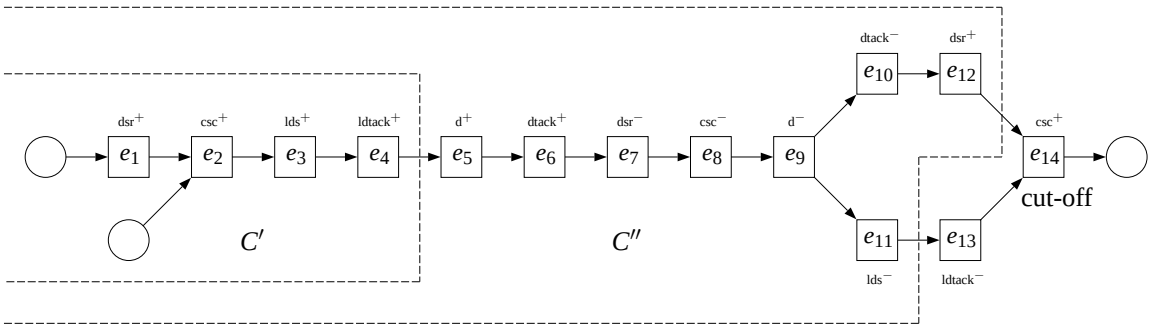
Suppose now that $X$ is one of the chosen supports of $z$. In order to derive an equation expressing $Nxt_z$ as a function of the signals in $X$, we build a Boolean formula $\mathcal{EQN}^z_X$ which has a variable $\mathsf{code}_x$ for each signal $x \in X$ and is satisfiable iff these variables can be assigned values in such a way that there is a reachable state $M$ such that $Code_x(M) = \mathsf{code}_x$, for all $x \in X$. Now, using the incremental SAT approach one can compute the projection of the set of reachable encodings onto $X$ (differentiating the stored solutions according to the value of $Nxt_z(M)$), and feed the result to a Boolean minimizer.

To summarize, the proposed method is executed separately for each signal $z \in Z_O$ and has three main stages: (I) computing the set $\mathcal{NSUPP}^z_{\max}$ of maximal non-supports of $z$; (II) computing the set $\mathcal{SUPP}^z_{\min}$ of minimal supports of $z$; and (III) deriving an equation for a chosen support $X$ of $z$. In the sequel, we describe each of these three stages in more detail.

It should be noted that the size of the truth table for Boolean minimization and the number of times a SAT solver is executed in our method can be exponential in the number of signals in the support. Thus, it is crucial for the performance of the proposed algorithm that the support of each signal 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 [2, 3].) This means that if an output signal 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 *technology mapping* step in the design cycle for asynchronous circuits (see, e.g., [3]); we do not consider them in this paper.

### 4.2. Computing maximal non-supports

Suppose that we want to compute the set of all maximal non-supports of a signal $z \in Z_O$. At the level of a branching process, a $CSC^z_X$ conflict can be represented as an unordered *conflict pair* of configurations $\langle C', C'' \rangle$ whose final states are in $CSC^z_X$ conflict, as shown in Figure 2.




| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| conf′ | = | 1111000000000 | | | | conf″ | = | 1111111111110 |
| en′ | = | 00001000000000 | nsupp | = | 110000 | en″ | = | 00000000000010 |
| code′ | = | 110101 | | | | code″ | = | 110000 |

**Figure 2. An unfolding prefix of the STG shown in Figure 1(a) illustrating a $CSC^{csc}_{\{dsr,ldtack\}}$ conflict between configurations $C'$ and $C''$. Note that $e_{14}$ is not enabled by $C''$ (since $e_{13} \notin C''$), and thus $Nxt_{csc}(C') = 1 \neq Nxt_{csc}(C'') = 0$. The order of signals in the binary encodings is: *dsr, ldtack, dtack, lds, d, csc*.**

We adopt the following naming conventions. The variable names are in the lower case and names of formulae are in the upper case. Names with a single prime (e.g., $\mathsf{conf}'_e$ and $\mathcal{CONF}'$) are related to $C'$, and ones with a double prime (e.g., $\mathsf{conf}''_e$) 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., $\mathsf{conf}'_e$ and $\mathsf{conf}''_e$ 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., $\mathsf{conf}'$ will denote the list of variables $\mathsf{conf}'_e$, where $e$ runs over the set $E \setminus E_{cut}$.

The following Boolean variables will be used in the proposed translation:

- For each event $e \in E \setminus E_{cut}$, we create two Boolean variables, $\mathsf{conf}'_e$ and $\mathsf{conf}''_e$, tracing whether $e \in C'$ and $e \in C''$, respectively.
- For each signal $x \in Z$, we create two Boolean variables, $\mathsf{code}'_x$ and $\mathsf{code}''_x$, tracing the values of $Code_x(C')$ and $Code_x(C'')$ respectively, and a variable $\mathsf{nsupp}_x$ indicating whether $x$ belongs to a non-support.
- For each condition $b \in B \setminus E^{\bullet}_{cut}$ such that $h(b) \in P^1_Z$, we create two Boolean variables, $\mathsf{cut}'_b$ and $\mathsf{cut}''_b$, tracing whether $b \in Cut(C')$ and $b \in Cut(C'')$ respectively.
- For each event $e \in E$ labelled by $z$, we create two Boolean variables, $\mathsf{en}'_e$ and $\mathsf{en}''_e$, tracing whether $e$ is 'enabled' by $C'$ and $C''$ respectively. Note that unlike $\mathsf{conf}'$ and $\mathsf{conf}''$, such variables are also created for the cut-off events.

As already mentioned, our aim is to build a Boolean formula $\mathcal{CSC}^z$ such that $Proj^{\mathcal{CSC}^z}_{\mathsf{nsupp}} = \mathcal{NSUPP}^z$, i.e., after assigning arbitrary values to the variables nsupp, the resulting formula is satisfiable iff there is a $CSC^z_X$ conflict, where $X \stackrel{\text{df}}{=} \{x \mid \mathsf{nsupp}_x = 1\}$. Figure 2 shows the satisfying assignment (except the variables $\mathsf{cut}'$ and $\mathsf{cut}''$) corresponding to the $CSC^{csc}_{\{dsr,ldtack\}}$ conflict depicted there. The target formula $\mathcal{CSC}^z$ 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 both legal configurations of the prefix (not just arbitrary sets of events). $\mathcal{CONF}'$ is defined as the conjunction of the formulae

$$\bigwedge_{e \in E \setminus E_{cut}} \bigwedge_{f \in {}^{\bullet}({}^{\bullet}e)} (\mathsf{conf}'_e \Rightarrow \mathsf{conf}'_f)$$

and

$$\bigwedge_{e \in E \setminus E_{cut}} \bigwedge_{f \in E_e} \neg(\mathsf{conf}'_e \wedge \mathsf{conf}'_f)\ ,$$

where $E_e \stackrel{\text{df}}{=} ((^\bullet e)^\bullet \setminus \{e\}) \setminus E_{cut}$. The former formula ensures that $C'$ is downward closed w.r.t. $\preceq$. The latter one ensures that $C'$ contains no structural conflicts. (One should be careful to avoid duplication of clauses when generating this formula.)

$\mathcal{CONF}'$ and $\mathcal{CONF}''$ can be transformed into the CNF by applying the rules $x \Rightarrow y \equiv \neg x \vee y$ and $\neg(x \wedge y) \equiv \neg x \vee \neg y$.

**Encoding constraint**

The role of this constraint is to ensure that $Code_x(C') = Code_x(C'')$ whenever $\mathsf{nsupp}_x = 1$. To build a formula establishing the value $\mathsf{code}'_x$ of each signal $x \in Z$ at the final state of $C'$, we observe that $\mathsf{code}'_x = 1$ 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). The latter can be captured by the constraint:

$$\bigwedge_{x \in Z} (\mathsf{code}'_x \iff \bigvee_{b \in B_x} \mathsf{cut}'_b) \,,$$

where $B_x \stackrel{\text{df}}{=} \{B \setminus E^\bullet_{cut} \mid h(b) = p^1_x\}$. We then define $\mathcal{CODE}'$ as the conjunction of the last formula and

$$\bigwedge_{x \in Z} \bigwedge_{b \in B_x} (\mathsf{cut}'_b \iff \bigwedge_{e \in {}^\bullet b} \mathsf{conf}'_e \wedge \bigwedge_{e \in b^\bullet \setminus E_{cut}} \neg \mathsf{conf}'_e) \,,$$

which ensures 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.) One can see that if $C'$ is a configuration and $\mathcal{CODE}'$ is satisfied then the value of signal $x$ at the final state of $C'$ is given by $\mathsf{code}'_x$. It is straightforward to build the CNF of $\mathcal{CODE}'$:

$$\bigwedge_{x \in Z} \Bigg( (\neg \mathsf{code}'_x \vee \bigvee_{b \in B_x} \mathsf{cut}'_b) \wedge \bigwedge_{b \in B_x} (\mathsf{code}'_x \vee \neg \mathsf{cut}'_b) \wedge \bigwedge_{b \in B_x} \Bigg( \bigwedge_{e \in {}^\bullet b} (\neg \mathsf{cut}'_b \vee \mathsf{conf}'_e) \wedge \bigwedge_{e \in b^\bullet \setminus E_{cut}} (\neg \mathsf{cut}'_b \vee \neg \mathsf{conf}'_e) \wedge (\mathsf{cut}'_b \vee \bigvee_{e \in {}^\bullet b} \neg \mathsf{conf}'_e \vee \bigvee_{e \in b^\bullet \setminus E_{cut}} \mathsf{conf}'_e) \Bigg) \Bigg) \,.$$

Moreover, $\mathcal{CODE}''$ and its CNF are built similarly.

Now we need a ensure that $\mathsf{code}'_x = \mathsf{code}''_x$ whenever $\mathsf{nsupp}_x = 1$. This can be expressed by the constraint $\mathcal{SUPP}$ defined as

$$\bigwedge_{x \in Z} \Big( \mathsf{nsupp}_x \Rightarrow (\mathsf{code}'_x \iff \mathsf{code}''_x) \Big) \,,$$

with the CNF

$$\bigwedge_{x \in Z} \Big( (\neg \mathsf{code}'_x \vee \mathsf{code}''_x \vee \neg \mathsf{nsupp}_x) \wedge (\mathsf{code}'_x \vee \neg \mathsf{code}''_x \vee \neg \mathsf{nsupp}_x) \Big) \,.$$

Now the encoding constraint can be expressed as $\mathcal{CODE}' \wedge \mathcal{CODE}'' \wedge \mathcal{SUPP}$.

**Next-state constraint**

The role of this constraint is to ensure that $Nxt_z(C') \neq Nxt_z(C'')$. Since all the other constraints are symmetric w.r.t. $C'$ and $C''$, one can rewrite it as $Nxt_z(C') = 0 \wedge Nxt_z(C'') = 1$. Moreover, it follows from the definition of $Nxt_z$ that $Nxt_z(C) \equiv \neg Code_z(C) \iff Out_z(C)$, and so the next-state constraint can be rewritten as the conjunction of $Code_z(C') \iff Out_z(C')$ and $\neg Code_z(C'') \iff Out_z(C'')$.

We observe that $z \in Z_O$ is enabled by $Mark(C')$ iff there is a $z^+$- or $z^-$–labelled event $e \notin C'$ 'enabled' by $C'$, i.e., such that $C' \cup \{e\}$ is a configuration (note that $e$ can be a cut-off event). We then define the formula $\mathcal{NEXTZERO}'$, ensuring that $Nxt_z(C') = 0$, as the conjunction of

$$\mathsf{code}'_z \iff \bigvee_{e \in E_z} \mathsf{en}'_e$$

and

$$\bigwedge_{e \in E_z} (\mathsf{en}'_e \iff \bigwedge_{f \in {}^\bullet({}^\bullet e)} \mathsf{conf}'_f \wedge \bigwedge_{f \in ({}^\bullet e)^\bullet \setminus E_{cut}} \neg \mathsf{conf}'_f) \,,$$

where $E_z \stackrel{\text{df}}{=} \{e \in E \mid \lambda(h(e)) \in \{z^+, z^-\}\}$. Intuitively, the former conjunct conveys that $Code_z(C') \iff Out_z(C)$ (it takes into account that $z$ is enabled by the final state of $C'$ iff at least one its instance is enabled by $C'$) and the latter one states for each instance $e$ of $z$ that $e$ is enabled by $C'$ iff all the events 'producing' tokens in ${}^\bullet e$ are in $C'$ but no events 'consuming' tokens from ${}^\bullet e$ (including $e$ itself) are in $C'$.

The formula $\mathcal{NEXTONE}''$, ensuring that $Nxt_z(C'') = 1$, is defined as the conjunction of

$$\neg \mathsf{code}''_z \iff \bigvee_{e \in E_z} \mathsf{en}''_e$$

and a constraint 'computing' $\mathsf{en}''_e$, which is similar to that for $\mathcal{NEXTZERO}'$. Now the next-state constraint can be expressed as $\mathcal{NEXTZERO}' \wedge \mathcal{NEXTONE}''$.

The CNF of $\mathcal{NEXTZERO}'$ is

$$(\neg \mathsf{code}'_z \vee \bigvee_{e \in E_z} \mathsf{en}'_e) \wedge \bigwedge_{e \in E_z} (\mathsf{code}'_z \vee \neg \mathsf{en}'_e) \wedge \bigwedge_{e \in E_z} \Bigg( \bigwedge_{f \in {}^\bullet({}^\bullet e)} (\neg \mathsf{en}'_e \vee \mathsf{conf}'_f) \wedge \bigwedge_{f \in ({}^\bullet e)^\bullet \setminus E_{cut}} (\neg \mathsf{en}'_e \vee \neg \mathsf{conf}'_f) \wedge (\mathsf{en}'_e \vee \bigvee_{f \in {}^\bullet({}^\bullet e)} \neg \mathsf{conf}'_f \vee \bigvee_{f \in ({}^\bullet e)^\bullet \setminus E_{cut}} \mathsf{conf}'_f) \Bigg) \,,$$

and the CNF of $\mathcal{NEXTONE}''$ can be built similarly.

**Translation to SAT**

The problem of computing the set $\mathcal{NSUPP}^z_{\max}$ of maximal non-supports of $z$ can now be formulated as a problem of finding the maximal elements of the projection $Proj^{\mathcal{CSC}^z}_{\mathsf{nsupp}}$ for the Boolean formula

$$\mathcal{CSC}^z \stackrel{\text{df}}{=} \mathcal{CONF}' \wedge \mathcal{CONF}'' \wedge \mathcal{CODE}' \wedge \mathcal{CODE}'' \wedge \mathcal{SUPP} \wedge \mathcal{NEXTZERO}' \wedge \mathcal{NEXTONE}'' .$$

It can be solved using the incremental SAT approach, as described in Section 3. Note that the size of this formula is polynomial in the size of the prefix (though new clauses are added during the incremental SAT run).

### 4.3. Computing minimal supports

Let $\mathcal{NSUPP}^z_{\max}$ be the set of maximal non-supports computed in the first stage of the method. Now we need to compute the set $\mathcal{SUPP}^z_{\min}$ of the minimal supports of $z$. This can be achieved by computing the set of minimal assignments for the Boolean formula

$$\bigwedge_{\mathsf{nsupp}^* \in \mathcal{NSUPP}^z_{\max}} \left( \bigvee_{x \in Z : \mathsf{nsupp}^*_x = 0} \mathsf{supp}_x \right),$$

which is satisfied by an assignment $A$ iff for all non-supports $\mathsf{nsupp}^* \in \mathcal{NSUPP}^z_{\max}$, $A \not\leqslant \mathsf{nsupp}^*$. This again can be done using the incremental SAT approach, as described in Section 3. Note that this Boolean formula is much smaller than that for the first stage of the method (it contains at most $|Z|$ variables), and thus the corresponding incremental SAT problem is much simpler.

### 4.4. Derivation of an equation

Suppose that $X$ is a (not necessarily minimal) support of $z$. We need to express $Nxt_z$ as a Boolean function of signals in $X$. This can be done by generating a truth table for $z$, similar to that shown in Figure 1(c) but with the first column restricted to signals in $X$, and then applying Boolean minimization.

The set of encodings appearing in the first column of the truth table coincides with the projections of the formula

$$\mathcal{EQN}^z_X \stackrel{\text{df}}{=} \mathcal{CONF}' \wedge \mathcal{CODE}'_X$$

onto the set of variables $\{\mathsf{code}_x \mid x \in X\}$, where $\mathcal{CODE}'_X$ is $\mathcal{CODE}'$ restricted to the set of signals $X$ (i.e., all the conjunctions of the form $\bigwedge_{x \in Z} \ldots$ are replaced by $\bigwedge_{x \in X} \ldots$). It also can be computed using the incremental SAT approach, as described in Section 3. Note that at each step of this computation, the SAT solver returns information not only about the next element of the projection, but also the values of all the other variables in the formula. That is, along with the restriction of some reachable encoding onto the set $X$ we have an information about a configuration $C$ via which it can be reached. Thus, the value of $Nxt_z$ on this element of the projection can be computed simply as $Nxt_z(C)$. This essentially completes the description of our method.

### 4.5. Optimizations

In the full version [8] of this paper, we describe optimizations which can significantly reduce the computation effort required by our method. First, we suggest a heuristic helping to compute a part of a signal's support without running the 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 $z$. (The information about triggers can be derived from the finite and complete prefix.) Then we show how to speed up the computation in the case of prefixes without structural conflicts. (The latter optimization is a straightforward generalization of that described in [7].)

## 5. Experimental results and conclusions

We implemented our method using the ZCHAFF SAT solver [12], and the benchmarks from [7] with modifications ensuring the CSC property and semi-modularity were attempted. All the experiments were conducted on a PC with a *Pentium*$^{TM}$ IV/2.8GHz processor and 512M RAM.

The first group of examples comes from real design practice. They are as follows:

- LAZYRINGCSC and RINGCSC — Asynchronous Token Ring Adapters described in [1, 9]. These two benchmarks were obtained from the LAZYRING and RING examples used in [7] by resolving CSC conflicts.
- DUP4PHCSC, DUP4PHMTRCSC and DUPMTRMODCSC — control circuits for the Power-Efficient Duplex Communication System described in [6]. These are the benchmarks from the corresponding series used in [7] which have CSC.
- CFSYMCSCA, CFSYMCSCB, CFSYMCSCC, CFSYMCSCD, CFASYMCSCA and CFASYMCSCB — control circuits for the Counterflow Pipeline Processor described in [15]. These are the same benchmarks as in [7].

Some of these STGs, although built by hand, are quite large in size.

Two other groups, PPWKCSC$(m,n)$ and PPARBCSC$(m,n)$, contain scalable examples of STGs modelling $m$ pipelines weakly synchronized without arbitration (in PPWKCSC$(m,n)$) and with arbitration (in PPARBCSC$(m,n)$).

| Problem | Net | | | | Prefix | | | Eqns | Time, [s] | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $\|S\|$ | $\|T\|$ | $\|Z_I\|/\|Z_O\|$ | $\|[M_0\rangle\|$ | $\|B\|$ | $\|E\|$ | $\|E_{cut}\|$ | (SAT) | PFY | SAT |
| *Real-Life STGs* | | | | | | | | | | |
| LAZYRING | 42 | 37 | 5/7 | 187 | 88 | 71 | 5 | 14 | 1 | <1 |
| RING | 185 | 172 | 11/18 | 16320 | 650 | 484 | 55 | 63 | 850 | 3 |
| DUP4PHCSC | 135 | 123 | 12/15 | 171 | 146 | 123 | 11 | 48 | 20 | <1 |
| DUP4PHMTRCSC | 114 | 105 | 10/16 | 149 | 122 | 105 | 8 | 46 | 13 | <1 |
| DUPMTRMODCSC | 152 | 115 | 10/17 | 321 | 228 | 149 | 13 | 165 | 125 | 1 |
| CFSYMCSCA | 85 | 60 | 8/14 | 6672 | 1341 | 720 | 56 | 60 | 163 | 16 |
| CFSYMCSCB | 55 | 32 | 8/8 | 690 | 160 | 71 | 6 | 34 | 10 | <1 |
| CFSYMCSCC | 59 | 36 | 8/10 | 2416 | 286 | 137 | 10 | 18 | 13 | <1 |
| CFSYMCSCD | 45 | 28 | 4/10 | 414 | 120 | 54 | 6 | 16 | 3 | <1 |
| CFASYMCSCA | 128 | 112 | 8/26 | 147684 | 1808 | 1234 | 62 | 450 | 1448 | 48 |
| CFASYMCSCB | 128 | 112 | 8/24 | 147684 | 1816 | 1238 | 62 | 93 | 2323 | 17 |
| *Marked Graphs* | | | | | | | | | | |
| PPWKCSC(2,3) | 24 | 14 | 0/7 | $2^7 = 128$ | 38 | 20 | 1 | 7 | <1 | <1 |
| PPWKCSC(2,6) | 48 | 26 | 0/13 | $2^{13} = 8192$ | 110 | 56 | 1 | 13 | 4 | <1 |
| PPWKCSC(2,9) | 72 | 38 | 0/19 | $2^{19} > 5 \cdot 10^5$ | 218 | 110 | 1 | 19 | 44 | <1 |
| PPWKCSC(2,12) | 96 | 50 | 0/25 | $2^{25} > 3 \cdot 10^7$ | 362 | 182 | 1 | 25 | 2082 | <1 |
| PPWKCSC(3,3) | 36 | 20 | 0/10 | $2^{10} = 1024$ | 57 | 29 | 1 | 10 | 1 | <1 |
| PPWKCSC(3,6) | 72 | 38 | 0/19 | $2^{19} > 5 \cdot 10^5$ | 165 | 83 | 1 | 19 | 43 | <1 |
| PPWKCSC(3,9) | 108 | 56 | 0/28 | $2^{28} > 2 \cdot 10^8$ | 327 | 164 | 1 | 28 | 7380 | <1 |
| PPWKCSC(3,12) | 144 | 74 | 0/37 | $2^{37} > 10^{11}$ | 543 | 272 | 1 | 37 | *time* | 1 |
| *STGs with Arbitration* | | | | | | | | | | |
| PPARBCSC(2,3) | 48 | 32 | 2/13 | $207 \cdot 2^4 = 3312$ | 110 | 66 | 2 | 18 | 4 | <1 |
| PPARBCSC(2,6) | 72 | 44 | 2/19 | $207 \cdot 2^{10} > 2 \cdot 10^5$ | 218 | 120 | 2 | 24 | 42 | <1 |
| PPARBCSC(2,9) | 96 | 56 | 2/25 | $207 \cdot 2^{16} > 10^7$ | 362 | 192 | 2 | 30 | 315 | <1 |
| PPARBCSC(2,12) | 120 | 68 | 2/31 | $207 \cdot 2^{22} > 8 \cdot 10^8$ | 542 | 282 | 2 | 36 | 3840 | 1 |
| PPARBCSC(3,3) | 71 | 48 | 3/19 | $297 \cdot 2^8 = 76032$ | 118 | 114 | 3 | 29 | 45 | <1 |
| PPARBCSC(3,6) | 107 | 66 | 3/28 | $297 \cdot 2^{17} > 3 \cdot 10^7$ | 368 | 204 | 3 | 38 | 1001 | <1 |
| PPARBCSC(3,9) | 143 | 84 | 3/37 | $297 \cdot 2^{26} > 10^{10}$ | 602 | 321 | 3 | 47 | 24941 | 1 |
| PPARBCSC(3,12) | 179 | 102 | 3/46 | $297 \cdot 2^{35} > 10^{13}$ | 890 | 465 | 3 | 56 | *mem* | 2 |

**Table 2. Experimental results.**

They are the benchmarks from the corresponding series used in [7] which satisfy the CSC property, with the latter series modified by 'factoring out' the arbiter into the environment to ensure semi-modularity. Note that in these two series of benchmarks all the signals except the arbiter's grants in PPARBCSC($m,n$) 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 a handshake (sometimes with an inverter attached to the output if the environment acts as an active port).

The experimental results are summarized 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 equations obtained by our method (this is equal to the total number of minimal supports for all the output signals and gives a rough idea of the explored design space); the time spent by the PETRIFY tool; and the time spent by the method proposed in this paper. We use 'mem' if there was a memory overflow and 'time' to indicate that the test had not stopped after 15 hours. We have not included in the table the time needed to build complete prefixes, since it did not exceed 0.1sec for each 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 unfolding-based method has a clear advantage over that based on state graphs.

Although the performed testing was limited in scope, we can draw some conclusions about the performance of the proposed algorithm. In all cases the proposed method 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 — it took less than 50 seconds to solve the hardest one.

It is important to note that these improvements in memory and running time come without any reduction in quality of the solutions. In fact, our method is *complete*, i.e., it can produce all the valid complex-gate implementations of each signal. However, in our implementation we restricted the algorithm to only minimal supports. Nevertheless, the explored design space was quite satisfactory: as the 'Eqns' column in Table 2 shows, in many cases our method proposed quite a few alternative implementations for a signal. In fact, among the list of solutions produced by our tool there was always a solution produced by PETRIFY (with, perhaps, only minor differences due to the non-uniqueness of the result of Boolean minimization). Overall, the proposed approach turned out to be clearly superior, especially for hard problem instances.

To conclude, according to the experimental results, the new method can solve quite large problem instances in relatively short time. It should also be emphasized that the unfolding approach is particularly well-suited for analyzing 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 [7, 10, 13] they form a complete design flow for complex-gate synthesis of asynchronous circuits based on STG unfolding prefixes rather than state graphs. In future work we intend to include also the technology mapping step into this framework.

## References


[1] C. Carrion and A. Yakovlev: Design and Evaluation of Two Asynchronous Token Ring Adapters. TRep. CS-TR-562, School of Comp. Science, Univ. of Newcastle (1996).

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

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

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

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

[6] S. B. Furber, A. Efthymiou and M. Singh: A Power-Efficient Duplex Communication System. Proc. of *AINT'00*, TU Delft, The Netherlands (2000) 145–150.

[7] V. Khomenko, M. Koutny and A. Yakovlev: Detecting State Coding Conflicts in STG Unfoldings Using SAT. Proc. of *ICACSD'03*, IEEE Comp. Soc. Press (2003) 51–60. Full version: to appear in Special Issue on Best Papers from ICACSD'2003, Fundamenta Informaticae.

[8] V. Khomenko, M. Koutny and A. Yakovlev: Logic Synthesis Avoiding State Space Explosion. TRep. CS-TR-813, School of Comp. Science, Univ. of Newcastle (2003). URL: `http://homepages.cs.ncl.ac.uk/victor.khomenko/home.formal/papers/papers.html`.

[9] K. S. Low and A. Yakovlev: Token Ring Arbiters: an Exercise in Asynchronous Logic Design with Petri Nets. TRep. CS-TR-537, School of Comp. Science, Univ. of Newcastle (1995).

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

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

[12] S. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik: CHAFF: Engineering an Efficient SAT Solver. Proc. of *DAC'01*, ASME Technical Publishing (2001) 530–535.

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

[14] A. Yakovlev, L. Lavagno and A. Sangiovanni-Vincentelli: A Unified Signal Transition Graph Model for Asynchronous Control Circuit Synthesis. *FMSD* 9(3) (1996) 139–188.

[15] A. Yakovlev: Designing Control Logic for Counterflow Pipeline Processor Using Petri nets. *FMSD* 12(1) (1998) 39–71.

[16] L. Zhang and S. Malik: The Quest for Efficient Boolean Satisfiability Solvers. Proc. of *CAV'02*, Springer-Verlag, LNCS 2404 (2002) 17–36.