Logic Synthesis Avoiding State Space Explosion

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. Following the description of our method, we present some problem-specific optimization rules. 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.
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 [3,6,27]. 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 (logic synthesis). One of the commonly used STG-based synthesis tools, Petrify [5,6], performs all of these steps automatically, after first constructing the reachability graph (in the form of a BDD [1]) of the initial STG specification. A vivid example of its use was the design of many circuits for the Amulet-3 microprocessor. Since popularity of this tool is steadily growing, it is very 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, had been applied for circuit synthesis. In [13, 14], 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 [3]. 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 either feasibility of a system of integer constraints [13] or satisfiability of a boolean formula (SAT) [14]. Those 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 [18], 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-gate synthesis that does not involve building reachability graphs at any stage. 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.

2 Basic definitions

In this section, we first present basic definitions concerning Petri nets and STGs, and then recall notions related to net unfoldings (see also [4–6,9,12,15,16,19,20,23,24,27]) and boolean satisfiability (see also [21,28]).

2.1 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., Figure 1(a)). If this hidden place contained a token, it is drawn directly on the arc. As usual, we will denote ${}^\bullet z \stackrel{\text{df}}{=} \{y \mid (y,z) \in F\}$ and $z^\bullet \stackrel{\text{df}}{=} \{y \mid (z,y) \in F\}$, for all $z \in P \cup T$, and ${}^\bullet Z \stackrel{\text{df}}{=} \bigcup_{z \in Z} {}^\bullet z$ and $Z^\bullet \stackrel{\text{df}}{=} \bigcup_{z \in Z} z^\bullet$, for all $Z \subseteq P \cup T$. We will assume that ${}^\bullet t \neq \varnothing \neq t^\bullet$, 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. 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 *executed*, leading to a marking M' given by $M' \stackrel{\text{df}}{=} M - {}^\bullet t + t^\bullet$, where '$-$' and '$+$' stand for the multiset difference and sum, respectively. 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 Σ 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 net system Σ is *k-bounded* if, for every reachable marking M and every place $p \in P$, $M(p) \leq k$, and *safe* if it is 1-bounded. Moreover, Σ is *bounded* if it is k-bounded for some $k \in \mathbb{N}$. One can show that the set $[M_0\rangle$ is finite iff Σ is bounded.

2.2 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 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

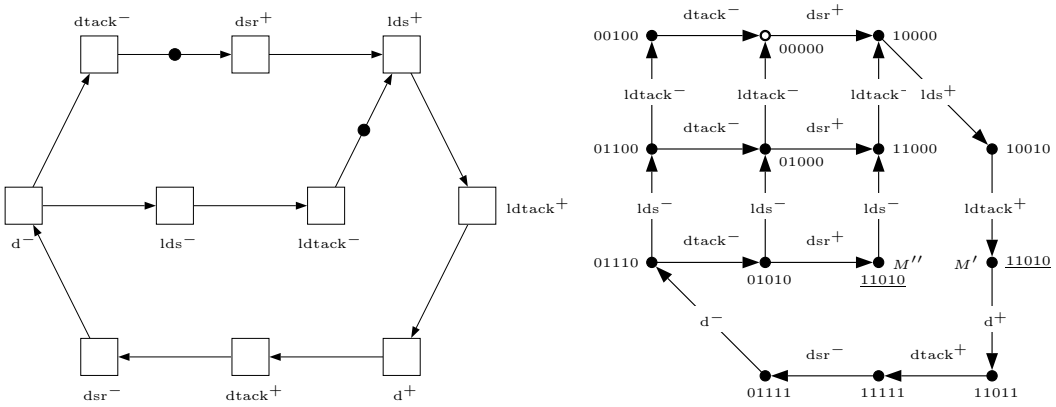


Fig. 1. An STG modelling a simplified VME bus controller **(a)** and its state graph with a CSC conflict between two states **(b)**. The order of signals in the binary encodings is: $dsr, ldtack, dtack, lds, d$.

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. We will 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 execution, reachable markings, and firing sequences.

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 corresponds to 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*[1] 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., $Code(M) \in \{0,1\}^{|Z|}$. Such a property guarantees that, for every signal $z \in Z$, the STG satisfies the following two conditions: (i) the first occurrence of z in the labelling of any firing sequence of Γ starting from M_0 has the same sign (either rising of falling); and (ii) the transitions corresponding to the rising and falling edges of z alternate in any firing sequence of Γ. In this paper it is assumed that all the STGs considered are consistent.[2] We will 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_z^0 and p_z^1, tracing the value of z as follows. Each z^+–labelled transition has p_z^0 in its preset and p_z^1 in its postset, and each z^-–labelled transition has p_z^1 in its preset and p_z^0 in its postset. 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

[1] This is a somewhat simplified notion of consistency; see [23] for a more elaborated one. Our approach works also for the notion presented there.

[2] The consistency of an STG can easily be checked during the process of building its finite and complete prefix [23].

(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 $P^0_Z \stackrel{\text{df}}{=} \{p^0_z \mid z \in Z\}$, $P^1_Z \stackrel{\text{df}}{=} \{p^1_z \mid z \in Z\}$, and $P_Z \stackrel{\text{df}}{=} P^0_Z \cup P^1_Z$.

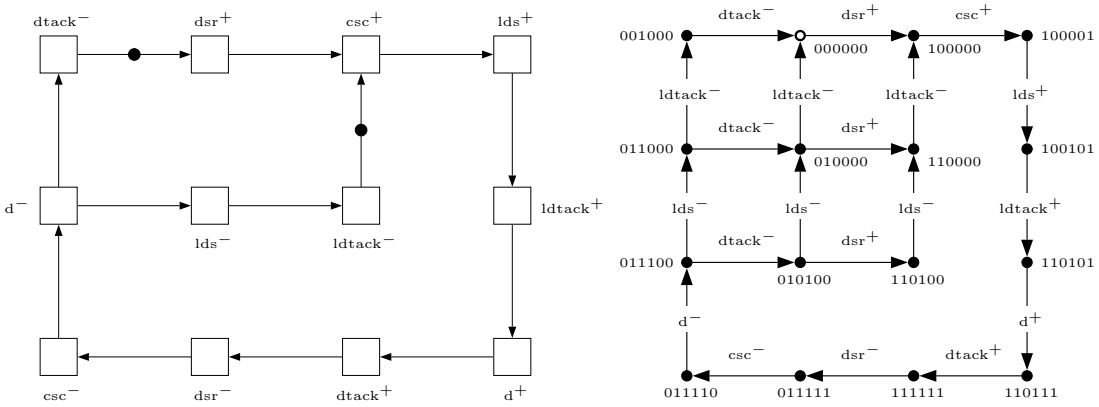
The *state graph* of Γ is a tuple $SG_\Gamma \stackrel{\text{df}}{=} (S, A, M_0, \mathit{Code})$ such that: $S \stackrel{\text{df}}{=} [M_0\rangle$ is the set of *states*; $A \stackrel{\text{df}}{=} \{M \stackrel{t}{\rightarrow} M' \mid M \in [M_0\rangle \wedge M[t\rangle M'\}$ is the set of *state transitions*; M_0 is the *initial state*; and $\mathit{Code} : S \rightarrow \{0,1\}^{|Z|}$ 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.

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 Γ as follows: $\mathit{Nxt}_z(M) \stackrel{\text{df}}{=} 0$ if $\mathit{Code}_z(M) = 0$ and no z^+–labelled transition is enabled at M, or $\mathit{Code}_z(M) = 1$ and a z^-–labelled transition is enabled at M; and $\mathit{Nxt}_z(M) \stackrel{\text{df}}{=} 1$ if $\mathit{Code}_z(M) = 1$ and no z^-–labelled transition is enabled at M, or $\mathit{Code}_z(M) = 0$ and a z^+–labelled transition is enabled at M. Moreover, the value of this function must be determined without ambiguity by the encoding of each reachable state, i.e., $\mathit{Nxt}_z(M)$ should be a function of $\mathit{Code}(M)$ rather than of M, i.e., $\mathit{Nxt}_z(M) = F_z(\mathit{Code}(M))$ 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_Γ, $z \in Z_O$ and $X \subseteq Z$. M' and M'' are in *Complete State Coding conflict for z w.r.t. X (CSC^z_X conflict)* if $\mathit{Code}_x(M') = \mathit{Code}_x(M'')$ for all $x \in X$ and $\mathit{Nxt}_z(M') \neq \mathit{Nxt}_z(M'')$. Γ satisfies the *CSC property for z (CSC^z property)* if no two states of SG_Γ are in CSC^z_Z conflict. Γ satisfies the *CSC property* if it satisfies the CSC^z property for each $z \in Z_O$.[3] X is a *support* of $z \in Z_O$ if no two states of Γ are in CSC^z_X conflict. In such a case the value of Nxt_z at each state M of SG_Γ 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 $Y \subset X$ is a support of z. In general, a signal can have several distinct minimal supports. Moreover, for each signal $z \in Z_O$ we define $\mathit{Out}_z(M)$ to be 1 if there exists a transition t enabled at M such that $\lambda(t) = z^{\pm}$; and 0 otherwise.

An example of an STG for a data read operation in a simple VME bus controller (a standard STG benchmark, see, e.g., [6]) is shown in Figure 1(a). It satisfies the CSC^{dtack} property (with the minimal support $\{d\}$), but not the CSC^d and CSC^{lds} properties. Part (b) of this figure illustrates a CSC conflict for these signals between two different states, M' and M'', that have the same encoding, 11010, but $\mathit{Nxt}_d(M') = 1 \neq \mathit{Nxt}_d(M'') = 0$ and $\mathit{Nxt}_{lds}(M') = 1 \neq \mathit{Nxt}_{lds}(M'') = 0$. This means that the values of $F_d(1,1,0,1,0)$ and $F_{lds}(1,1,0,1,0)$ are ill-defined (they should be 1 according to the state M' and 0 according to the state M''), and thus these signals are not implementable as a logic gates. To cope with this, an additional signal, *csc*, helping to resolve this CSC conflict is added to the STG, e.g., as shown in Figure 2(a,b). (Note that the circuit has to implement this new signal, and so for the purpose of logic synthesis it is regarded as output, though it is invisible to the environment.) Now the equations implementing each output signal can be obtained by applying boolean minimization to the truth table shown in Figure 2(c). 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 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

[3] This definition, though different in form from the conventional one (see, e.g., [13, 14]), is equivalent to it due to the fact that $\mathit{Nxt}_z(M') = \mathit{Nxt}_z(M'')$ for all $z \in Z_O$ iff the sets of output signals enabled at M' and M'' are the same, provided that $\mathit{Code}(M') = \mathit{Code}(M'')$.



(a) **(b)**

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

$Code(M)$	$Nxt_{dtack}(M)$	$Nxt_{lds}(M)$	$Nxt_{d}(M)$	$Nxt_{csc}(M)$
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
Expression:	d	$d \vee csc$	$csc \wedge ldtack$	$dsr \wedge (\neg ldtack \vee csc)$

(c)

Fig. 2. An STG **(a)** where the CSC conflict has been resolved by adding a new signal, *csc*; its state graph **(b)**; and the truth table for the output signals **(c)** with the last row showing the result of boolean minimization. The order of signals in the binary encodings is: *dsr*, *ldtack*, *dtack*, *lds*, *d*, *csc*.

can exploit this to reduce the complexity of the resulting boolean expression). The last row of the table gives the result of boolean minimization, viz. the expressions computed by logic gates implementing the output signals of the circuit. This essentially completes the standard complex gate synthesis procedure based on state graphs.

2.3 Branching processes

Two nodes of a net $N = (P, T, F)$, y and y', are in *structural conflict*, denoted by $y \# y'$, if there are distinct transitions $t, t' \in T$ such that ${}^\bullet t \cap {}^\bullet t' \neq \varnothing$ and (t, y) and (t', y') are in the reflexive transitive closure of the flow relation F, denoted by $\preceq$. A node y is in *structural self-conflict* if $y \# y$.

An *occurrence net* is a net $ON \stackrel{\text{df}}{=} (B, E, G)$ where B is the set of *conditions* (places), E is the set of *events* (transitions) and G is a *flow relation*. It is assumed that: ON is acyclic (i.e., $\preceq$ is a partial order); for every $b \in B$, $|{}^\bullet b| \leq 1$; for every $y \in B \cup E$, $\neg(y \# y)$ and there are finitely many y' such that $y' \prec y$, where $\prec$ denotes the irreflexive transitive closure of G. $Min(ON)$ will denote the minimal elements of $B \cup E$ with respect to $\preceq$. The relation $\prec$ is the *causality relation*. Two nodes are *concurrent*, denoted $y \; co \; y'$, if neither $y \# y'$ nor $y \preceq y'$ nor $y' \preceq y$.

A *homomorphism* from an occurrence net ON to a net system Σ is a mapping $h : B \cup E \to S \cup T$ such that: $h(B) \subseteq S$ and $h(E) \subseteq T$ (conditions are mapped to places, and events to transitions); for all $e \in E$, the restriction of h to ${}^\bullet e$ is a bijection between ${}^\bullet e$ and ${}^\bullet h(e)$ and the restriction of h to $e^\bullet$ is a bijection between $e^\bullet$ and $h(e)^\bullet$ (transition environments are preserved); the restriction of h to $Min(ON)$ is a bijection between $Min(ON)$ and M_0 (minimal conditions correspond to the initial marking); and for all $e, f \in E$, if ${}^\bullet e = {}^\bullet f$ and $h(e) = h(f)$ then $e = f$ (there is no redundancy).

A *branching process* of Σ is a quadruple $\pi \stackrel{\text{df}}{=} (B, E, G, h)$ such that (B, E, G) is an occurrence net and h is a homomorphism from it to Σ. A branching process $\pi' = (B', E', G', h')$ of Σ is a *prefix* of a branching process $\pi = (B, E, G, h)$, denoted $\pi' \sqsubseteq \pi$, if (B', E', G') is a subnet of (B, E, G) such that: if $e \in E'$ and $(b, e) \in G$ or $(e, b) \in G$ then $b \in B'$; if $b \in B'$ and $(e, b) \in G$ then $e \in E'$; and h' is the restriction of h to $B' \cup E'$. For each Σ there exists a unique (up to isomorphism) maximal (w.r.t. $\sqsubseteq$) branching process, called the *unfolding* of Σ (it is infinite whenever Σ has an infinite execution).

2.4 Configurations and cuts

A *configuration* of an occurrence net ON is a set of events C such that for all $e, f \in C$, $\neg(e \# f)$ and, for every $e \in C$, $f \prec e$ implies $f \in C$. The configuration $[e] \stackrel{\text{df}}{=} \{f \mid f \preceq e\}$ is called the *local configuration* of $e \in E$, and $\langle e \rangle \stackrel{\text{df}}{=} [e] \setminus \{e\}$ denotes the set of *causal predecessors* of e. A *cut* is a maximal (w.r.t. $\subseteq$) set of conditions B' such that $b \; co \; b'$, for all distinct $b, b' \in B'$. Every marking reachable from $Min(ON)$ is a cut.

Let C be a finite configuration of a branching process π. Then $Cut(C) \stackrel{\text{df}}{=} (Min(ON) \cup C^\bullet) \setminus {}^\bullet C$ is a cut; moreover, the multiset of places $h(Cut(C))$ is a reachable marking of Σ, denoted $Mark(C)$. A marking M of Σ is *represented* in π if the latter contains a finite configuration C such that $M = Mark(C)$. Every marking represented in π is reachable, and every reachable marking is represented in the unfolding of Σ.

A branching process $\pi = (B, E, G, h)$ of Σ is *complete* if there is a set $E_{cut} \subseteq E$ of *cut-off* events such that for every reachable marking M of Σ there exist a finite configuration C of π such that $C \cap E_{cut} = \varnothing$ and $M = Mark(C)$, and for every transition t enabled by M, there is an event $e \notin C$ in π such that $h(e) = t$ and $C \cup \{e\}$ is a configuration (e may be a cut-off event).[4]

Although, in general, an unfolding is infinite, for every bounded net system Σ one can construct a finite complete prefix of the unfolding of Σ, by choosing an appropriate set E_{cut} of cut-off events, beyond which the unfolding is not generated.

A *branching process* of an STG $\Gamma = (\Sigma, Z, \lambda)$ is a branching process of Σ augmented with an additional labelling of its events, $\lambda \circ h : E \to Z^{\pm}$. One can easily check the consistency of Γ, once its finite and complete prefix has been built [23].

We also extend the functions $Code$, $Code_z$, Nxt_z, and Out_z to finite configurations of the branching process of Γ 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 $Out_z(C) \stackrel{\text{df}}{=} Out_z(Mark(C))$.

[4] This notion of completeness differs from the one given in [9], which does not mention cut-off events, and hence is not appropriate for algorithms making use of them. Having said that, one can show that the unfolding algorithm proposed in [9] builds prefixes which are complete not only in the sense of the definition given in [9], but also in the stronger sense assumed here.

2.5 Boolean satisfiability

The *boolean satisfiability (SAT) problem* consists in finding a *satisfying assignment*, i.e., a mapping $Var \to \{0,1\}$ defined on the set of variables Var occurring in a given boolean expression φ such that φ evaluates to 1. (Note that we identify the booleans **false** and **true** with integers 0 and 1, respectively, provided that this does not create confusion.) This expression is often assumed to be given in the *conjunctive normal form (CNF)*

$$\varphi = \bigwedge_{i=1}^{n} \bigvee_{l \in L_i} l \,,$$

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.

In order to solve a boolean satisfiability problem, SAT solvers perform exhaustive search assigning the values 0 or 1 to the variables. To reduce the search space, they use various heuristics (see, e.g., [28] for a brief overview). An almost universally used one is the *Boolean Constraint Propagation (BCP)* rule, which tells that if all but one literals occurring in some clause have the value 0 then in order to satisfy the clause the remaining literal must have the value 1. This rule is applied iteratively, until no more variables can be assigned, on each step of the search. If at some point all the literals in some clause have the value 0 then the built partial assignment cannot be a part of any satisfying assignment, and the solver backtracks.

Some of the leading SAT solvers, e.g., zChaff [21], 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 [28]) collected so far. In particular, such an approach can be used to compute *projections* of assignments satisfying a given formula, as described in sequel.

Projecting satisfying assignments 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.

Step 0: $\mathcal{A} = \varnothing$.
Step 1: Run the SAT solver for φ.
Step 2: If φ 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 φ 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.

Note that the procedure is correct since it terminates (as Step 4 eliminates at least one satisfying assignment, viz. the A found in Step 1) and returns $Proj_V^\varphi$ (as Step 4 eliminates only those satisfying assignments A' which have the same restriction $A'|_V = A|_V$).

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 φ by appending a new clause $\bigvee_{v \in V \wedge A(v)=1} \neg v$.

Moreover, before terminating, an additional pass over the elements stored in $\mathcal{A}$ is made in order to eliminate any non-minimal projections.

The modified procedure works since Step 4' eliminates at least one satisfying assignment, viz. the A found in Step 1 (notice that if the all-zeros is *the* minimal element of $Proj_V^\varphi$ then

Step 4' produces an unsatisfiable formula). Moreover, Step 4' never eliminates any minimal element of $Proj_V^{\varphi}$ other than $A|_V$ which has already been stored in $\mathcal{A}$.

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 φ by appending a new clause $\bigvee_{v \in V \wedge A(v)=0} v$.

And, before terminating, an additional pass over the elements stored in $\mathcal{A}$ would be made in order to eliminate any non-maximal projections. It is worth noting that the iterative procedure is usually fast on the initial iterations as the formula φ is typically easily satisfiable, but it may become harder towards the end of its run.

Note that a similar computation could be implemented using Binary Decision Diagrams (BDDs) [1], by eliminating quantifiers from $\exists(Var \backslash V)\varphi$ and then computing the solutions of the resulting formula. However, in practice, V is a small set (it corresponds to the inputs of a logic gate computing an output signal), and so such an approach would have to eliminate too many quantifiers, while the approach based on the incremental SAT benefits from this.

3 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 it [13, 14] 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. As a result, unfolding-based methods had clear advantage over the BDD-based techniques both in terms of memory usage and running time.

3.1 Outline of the proposed method

In [14], 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 $A|_{\mathsf{nsupp}}$. The key property of $\mathcal{CSC}^z$ is that $\mathcal{NSUPP}^z = Proj_{\mathsf{nsupp}}^{\mathcal{CSC}^z}$, 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 a support of z already chosen. In order to derive an equation expressing Nxt_z as a function of the signals in X, we build a boolean formula $\mathcal{EQN}_X$ 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 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 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 [3, 6].) 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., [6]); we do not consider them in this paper.

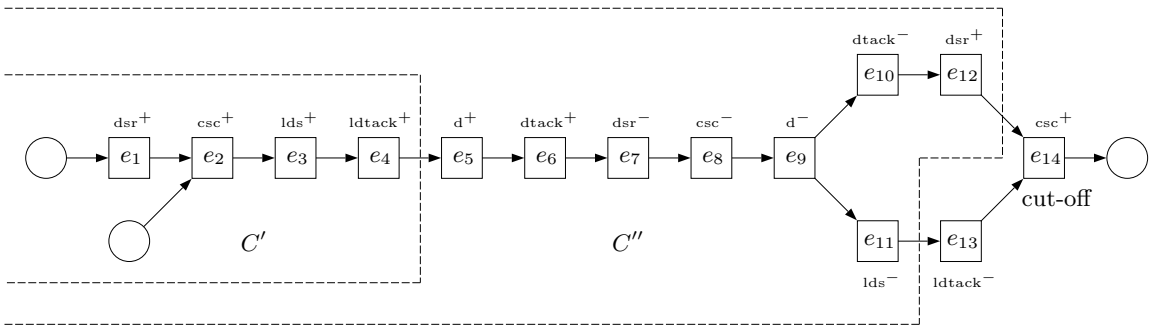
3.2 Computing maximal non-supports

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 3.

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., 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' will denote the list of variables conf'_e, where e runs over the set $E \setminus E_{cut}$.

The following boolean variables will be used in the proposed translation (Figure 3 shows the values of these variables for the depicted conflict pair of configurations):

- 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 two boolean variables, code'_x and code''_x, tracing the the values of $Code_x(C')$ and $Code_x(C'')$ respectively, and a variable nsupp_x indicating whether x belongs to a non-support.



$\mathsf{conf}' = 1111000000000$ $\mathsf{en}' = 00001000000000$ $\mathsf{code}' = 110101$

$\mathsf{nsupp} = 110000$

$\mathsf{conf}'' = 1111111111110$ $\mathsf{en}'' = 00000000000010$ $\mathsf{code}'' = 110000$

Fig. 3. An unfolding prefix of the STG shown in Figure 2**(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.

- For each condition $b \in B \setminus E^{\bullet}_{cut}$ such that $h(b) \in P^1_Z$, we create two boolean variables, cut'_b and 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, 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.

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 3 shows the satisfying assignment (except the variables cut' and cut'') corresponding to the CSC^z_X 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 following 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. $\preceq$, i.e., if $e \in C'$ then its immediate predecessors are also in C'. The latter one ensures that C' contains no structural conflicts. (One should be careful to avoid duplication of clauses when generating this formula.) Note that one can shorten this formulae by replacing ${}^\bullet({}^\bullet e)$ by $\max_{\preceq} {}^\bullet({}^\bullet e)$ and E_e by $\min_{\preceq} E_e$. $\mathcal{CONF}''$ is defined similarly.

$\mathcal{CONF}'$ 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 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 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 the final state of C' iff there is a $z^\pm$–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 $E_z \stackrel{\text{df}}{=} \{e \in E \mid \lambda(h(e)) = z^{\pm}\}$. The former conjunct ensures 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'. Note that one can shorten the latter formula by replacing ${}^\bullet({}^\bullet e)$ by $\max_{\preceq} {}^\bullet({}^\bullet e)$ and $({}^\bullet e)^\bullet \setminus E_{cut}$ by $\min_{\preceq}(({}^\bullet e)^\bullet \setminus E_{cut})$.

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' 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

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

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 projections $Proj^{\mathcal{CSC}^z}_{\mathsf{nsupp}}$ for the boolean formula

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

And it can be solved using the incremental SAT approach, as described in Section 2.5.

3.3 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 2.5. Note that the last 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.

3.4 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 2(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

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

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 2.5. 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 Optimizations

In this section, 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. 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 [13, 14].)

4.1 Simplifying support computation

As it was already noted, the number of solver runs in our method can be exponential in the size of a support of an output signal z. Thus it makes sense to find at least a part of the support using suitable heuristics.

We define for a $z^{\pm}$-labelled event e the set of its *triggers* as $Trig(e) \stackrel{\text{df}}{=} \max_{\prec}\langle e\rangle$. (Intuitively, $Trig(e)$ comprises those events whose firing can 'trigger' the firing of e.) We also define the set $Trig(z)$ as the set of signals whose instances can trigger an instance of z in the (full) unfolding.

One can show that $Trig(z)$ is a subset of any support of z. Indeed, firing a trigger x can change the 'enabledness status' of z, i.e., there are two states in the state graph with the encodings which differ only in the position corresponding to this trigger and such that the values of Nxt_z at them are different. That is, these two states are in $CSC^z_{Z\setminus\{x\}}$ conflict, and so any set of signals which does not contain x is a non-support.

Using this observation, one can simplify the first stage of the method by pre-setting the values of nsupp_x corresponding to the signals in $Trig(z)$ to 1 and simplifying the formula accordingly before running the solver.

It should be noted, however, that the set $Trig(z)$ was defined on the whole unfolding rather than its finite and complete prefix, i.e., it does not necessarily coincide with the set of signals whose instances can trigger an instance of z in such a prefix. Nevertheless, the latter set is an underapproximation of $Trig(z)$ and can still be used without affecting the correctness of the method.

4.2 The case of prefixes without structural conflicts

In many cases the performance of the proposed method can be improved by exploiting specific properties of the Petri net underlying an STG Γ. For instance, if Γ is free from dynamic choices (in particular, this is the case for STGs which are marked graphs) then the union of any two configurations of its unfolding is also a configuration. (Note that freeness from structural conflicts can easily be detected: it is enough to check that $|b^\bullet| \leq 1$, for all conditions b of the prefix.) This observation can be used to reduce the search space. Indeed, according to Proposition 1 below, it is then enough to look only for those cases when the configurations C' and C'' being tested are ordered in the set-theoretical sense.

Proposition 1. *Let $\langle C', C''\rangle$ be a CSC^z_X conflict pair of configurations in the unfolding of a consistent STG such that $C' \not\subseteq C''$, $C'' \not\subseteq C'$, and $C' \cup C''$ is a configuration. Then $\langle C, C'\rangle$ or $\langle C, C''\rangle$ is a CSC^z_X conflict pair, where $C \stackrel{\text{df}}{=} C' \cap C''$.*

Proof. Since $C' \cup C''$ is a configuration, each event in $C' \setminus C'' \neq \varnothing$ is concurrent to each event in $C'' \setminus C' \neq \varnothing$. Now, due to the consistency of the STG, no two distinct concurrent events in its unfolding can have the same signal label. Hence none of the events in $C' \setminus C''$ can have the same signal label (even after ignoring the sign) as an event in $C'' \setminus C'$. Consequently, since $Code_x(C') = Code_x(C'')$ for each $x \in X$, $Code_x(C') - Code_x(C) = Code_x(C'') - Code_x(C) = 0$, i.e., $Code_x(C) = Code_x(C') = Code_x(C'')$. Moreover, since $Nxt_z(C') \neq Nxt_z(C'')$, $Nxt_z(C)$ differs from at least one of $Nxt_z(C')$ and $Nxt_z(C'')$, i.e., $\langle C, C' \rangle$ or $\langle C, C'' \rangle$ is a CSC_X^z conflict pair.

In order to consider only ordered pairs of configurations, it is enough to add to the formula $\mathcal{CSC}^z$ constructed in the first stage of the method the constraint

$$\bigwedge_{e \in E \setminus E_{cut}} \Big((\mathsf{v}_{\subseteq} \Rightarrow (\mathsf{conf}'_e \Rightarrow \mathsf{conf}''_e)) \wedge (\neg \mathsf{v}_{\subseteq} \Rightarrow (\mathsf{conf}''_e \Rightarrow \mathsf{conf}'_e)) \Big) ,$$

requiring that $C' \subseteq C'' \vee C'' \subseteq C'$, where $\mathsf{v}_{\subseteq}$ is a new variable which can be set arbitrarily by the solver. This constraint can easily be transformed into the CNF by applying the rule $x \Rightarrow y \equiv \neg x \vee y$.

Note that because the next-state constraint is asymmetric, we cannot limit the search space by assuming that, say, $C' \subseteq C''$, and have to explore both possibilities.

5 Experimental results

We implemented our method using the zChaff SAT solver [21], and the benchmarks from [13, 14] with modifications ensuring the CSC property and semi-modularity were attempted. All the experiments were conducted on a PC with $Pentium^{TM}$ IV/2.8GHz processor and 512M RAM.

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

- LazyRingCsc and RingCsc — Asynchronous Token Ring Adapters described in [2, 17]. These two benchmarks were obtained from the LazyRing and Ring examples used in [13, 14] by resolving CSC conflicts.
- Dup4phCsc, Dup4phMtrCsc, and DupMtrModCsc — control circuits for the Power-Efficient Duplex Communication System described in [10]. These are the benchmarks from the corresponding series used [13, 14] which satisfy the CSC property.
- CfSymCscA, CfSymCscB, CfSymCscC, CfSymCscD, CfAsymCscA, and CfAsymCscB — control circuits for the Counterflow Pipeline Processor described in [26]. These are the same benchmarks as in [13, 14].

Some of these STGs, although built by hand, are quite large in size. The results for this group are summarized in the first part of Table 1.

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)). They are the benchmarks from the corresponding series used [13, 14] which satisfy the CSC property, with the latter series modified by 'factoring out' the arbiter 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 (sometimes with an inverter if the environment acts as an active port). Figure 4 illustrates these two types of STGs, and the results for these two groups are summarized in the last two parts of Table 1.

The meaning of the columns in Table 1 is as follows (from left to right): the name of the problem; the number of places, transitions, and input and output signals in the original STG; the number of conditions, events and cut-off events in the complete prefix; the total number of

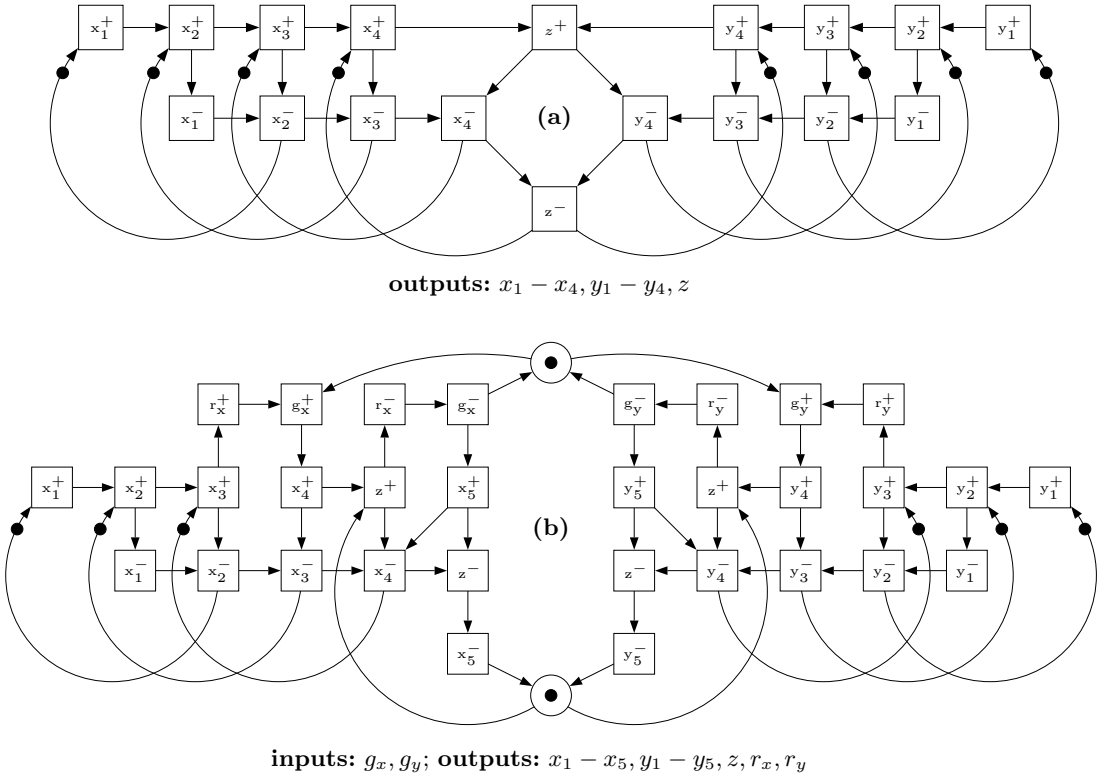


Fig. 4. An STG modeling two weakly synchronized pipelines without arbitration **(a)** and with arbitration **(b)**.

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 all the attempted STGs.

Note that in all cases the size of the complete prefix was relatively small. As already mentioned, 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. As a result, unfolding-based methods have a clear advantage over the state graph ones.

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. The explored design space also seems to be satisfactory: as the 'Eqns' column in Table 1 shows, in many cases our method proposed quite a few alternative implementations for a signal. Overall, the proposed approach turned out to be clearly superior, especially for hard problem instances.

Such an efficiency is due to the fact that the clauses comprising the formula are short (most of them contain only 2 or 3 literals) and thus allow for a good propagation of the variables' values during the application of the BCP rule by the SAT solver. Moreover, the BCP rule applied to, e.g., the configuration constraint $\mathcal{CONF}'$ captures the following dependencies (exploited

Problem	Net			Prefix			Eqns	Time, [s]	
	$\|S\|$	$\|T\|$	$\|Z_I\|/\|Z_O\|$	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	(SAT)	Pfy	Sat
Real-Life STGs									
LazyRing	42	37	5/7	88	71	5	14	1	<1
Ring	185	172	11/18	650	484	55	63	850	3
Dup4phCsc	135	123	12/15	146	123	11	48	20	<1
Dup4phMtrCsc	114	105	10/16	122	105	8	46	13	<1
DupMtrModCsc	152	115	10/17	228	149	13	165	125	1
CfSymCscA	85	60	8/14	1341	720	56	60	163	16
CfSymCscB	55	32	8/8	160	71	6	34	10	<1
CfSymCscC	59	36	8/10	286	137	10	18	13	<1
CfSymCscD	45	28	4/10	120	54	6	16	3	<1
CfAsymCscA	128	112	8/26	1808	1234	62	450	1448	48
CfAsymCscB	128	112	8/24	1816	1238	62	93	2323	17
Marked Graphs									
PpWkCsc(2,3)	24	14	0/7	38	20	1	7	<1	<1
PpWkCsc(2,6)	48	26	0/13	110	56	1	13	4	<1
PpWkCsc(2,9)	72	38	0/19	218	110	1	19	44	<1
PpWkCsc(2,12)	96	50	0/25	362	182	1	25	2082	<1
PpWkCsc(3,3)	36	20	0/10	57	29	1	10	1	<1
PpWkCsc(3,6)	72	38	0/19	165	83	1	19	43	<1
PpWkCsc(3,9)	108	56	0/28	327	164	1	28	7380	<1
PpWkCsc(3,12)	144	74	0/37	543	272	1	37	*time*	1
STGs with Arbitration									
PpArbCsc(2,3)	48	32	2/13	110	66	2	18	4	<1
PpArbCsc(2,6)	72	44	2/19	218	120	2	24	42	<1
PpArbCsc(2,9)	96	56	2/25	362	192	2	30	315	<1
PpArbCsc(2,12)	120	68	2/31	542	282	2	36	3840	1
PpArbCsc(3,3)	71	48	3/19	118	114	3	29	45	<1
PpArbCsc(3,6)	107	66	3/28	368	204	3	38	1001	<1
PpArbCsc(3,9)	143	84	3/37	602	321	3	47	24941	1
PpArbCsc(3,12)	179	102	3/46	890	465	3	56	*mem*	2

Table 1. Experimental results.

by a special-purpose integer programming solver in [13]) between the variables implied by the causal order on events: if conf'_e is 1 then the BCP rule will set each conf'_f such that $f \prec e$ to 1, and each conf'_g such that $g \# e$ to 0. Similarly, if conf'_e is 0 then, for each $f \succ e$, conf'_f will be set to 0.

It is also worth to investigate the break down of our method's execution time. It turns out that it spends a substantial amount of time (around 7%) generating SAT instances. This is because we have not yet addressed the optimization of generating the formulae, and we expect this time can be significantly improved. Stage (I) typically takes 25–80% of time, and stage (III) takes 10–50%. Stage (II) and boolean minimization take negligible amount of time (much less than 1%).

6 Conclusions

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, as it was already noted, 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 [14,18,23] 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.

An important observation one can make is that the combination 'unfolder & solver' is quite powerful. It has already been used in a number of papers (see, e.g., [11, 13]). Most of 'interesting' problems for safe Petri nets are $\mathcal{PSPACE}$-complete [8], and unfolding such a net allows to reduce this complexity class down to $\mathcal{NP}$ (or even $\mathcal{P}$ for some problems). Though the size of a finite and complete unfolding prefix can be exponential in the size of the original Petri net, in practice it is often relatively small. In particular, according to our experiments, this is almost always the case for STGs. A problem formulated for a prefix can usually be translated into some canonical problem, e.g., an integer programming one [13], a problem of finding a stable model of a logic program [11], or a boolean satisfiability problem as in this paper. Then an appropriate solver can be used for efficiently solving it.

Acknowledgements

This research was supported by an EPSRC grant GR/R16754 (BESST).

References

1. Bryant, R.: Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Transactions on Computers*, **C-35-8**, 1986, 677–691.
2. Carrion, C., Yakovlev, A.: *Design and Evaluation of Two Asynchronous Token Ring Adapters*, Technical Report CS-TR-562, Department of Computing Science, University of Newcastle upon Tyne, 1996.
3. Chu, T.-A.: *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*, Ph.D. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1987.
4. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: Complete State Encoding Based on Theory of Regions, *Proc. ASYNC'1996*, IEEE Computer Society Press, 1996.
5. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: PETRIFY: a Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers, *IEICE Transactions on Information and Systems*, **E80-D**(3), 1997, 315–325.
6. Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., Yakovlev, A.: *Logic Synthesis of Asynchronous Controllers and Interfaces*, Springer-Verlag, 2002.
7. Engelfriet, J.: Branching Processes of Petri Nets, *Acta Informatica*, **28**, 1991, 575–591.
8. Esparza, J.: Decidability and Complexity of Petri Net Problems — an Introduction, *Lectures on Petri Nets I: Basic Models* (W. Reisig, G. Rozenberg, Eds.), Lecture Notes in Computer Science 1491, Springer-Verlag, 1998.
9. Esparza, J., Römer, S., Vogler, W.: An Improvement of McMillan's Unfolding Algorithm, *Formal Methods in System Design*, **20**(3), 2002, 285–310.
10. Furber, S., Efthymiou, A., Singh, M.: A Power-Efficient Duplex Communication System, *Proc. AINT'2000* (A. Yakovlev, R. Nouta, Eds.), TU Delft, The Netherlands, 2000.
11. Heljanko, K.: Using Logic Programs with Stable Model Semantics to Solve Deadlock and Reachability Problems for 1-Safe Petri Nets, *Fundamenta Informaticae*, **37**(3), 1999, 247–268.
12. Khomenko, V., Koutny, M., Vogler, W.: Canonical Prefixes of Petri Net Unfoldings, *Proc. CAV'2002* (E. Brinksma, K. Larsen, Eds.), Lecture Notes in Computer Science 2404, Springer-Verlag, 2002, Full version: to appear in Acta Informatica.
13. Khomenko, V., Koutny, M., Yakovlev, A.: Detecting State Coding Conflicts in STGs Using Integer Programming, *Proc. DATE'2002* (C. Kloos, J. Franca, Eds.), IEEE Computer Society Press, 2002.
14. Khomenko, V., Koutny, M., Yakovlev, A.: Detecting State Coding Conflicts in STG Unfoldings Using SAT, *Proc. ICACSD'2003* (J. Lilius, F. Balarin, R. Machado, Eds.), IEEE Computer Society Press, 2003, Full version: to appear in Special Issue on Best Papers from ICACSD'2003, Fundamenta Informaticae.

15. Kondratyev, A., Cortadella, J., Kishinevsky, M., Lavagno, L., Taubin, A., Yakovlev, A.: Identifying State Coding Conflicts in Asynchronous System Specifications Using Petri Net Unfoldings, *Proc. ICACSD'1998*, IEEE Computer Society Press, 1998.
16. Kondratyev, A., Cortadella, J., Kishinevsky, M., Pastor, E., Roig, O., Yakovlev, A.: Checking Signal Transition Graph Implementability by Symbolic BDD Traversal, *Proc. DATE'1995*, IEEE Computer Society Press, 1995.
17. Low, K., Yakovlev, A.: *Token Ring Arbiters: an Exercise in Asynchronous Logic Design with Petri Nets*, Technical report, Department of Computing Science, University of Newcastle upon Tyne, 1995.
18. Madalinski, A., Bystrov, A., Khomenko, V., Yakovlev, A.: Visualization and Resolution of Coding Conflicts in Asynchronous Circuit Design, *Proc. DATE'2003*, IEEE Computer Society Press, 2003, Full version: to appear in Special Issue on Best Papers from DATE'2003, IEE Proceedings: Computers & Digital Techniques.
19. McMillan, K.: *Symbolic Model Checking: an Approach to the State Explosion Problem*, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, 1992.
20. McMillan, K.: A Technique of State Space Search Based on Unfoldings, *Formal Methods in System Design*, **6**(1), 1995, 45–65.
21. Moskewicz, S., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: CHAFF: Engineering an Efficient SAT Solver, *Proc. DAC'2001*, ASME Technical Publishing, 2001.
22. Pastor, E., Cortadella, J., Roig, O.: Symbolic Analysis of Bounded Petri Nets, *IEEE Transactions on Computers*, **50**(5), 2001, 432–448.
23. Semenov, A.: *Verification and Synthesis of Asynchronous Control Circuits Using Petri Net Unfolding*, Ph.D. Thesis, Department of Computing Science, University of Newcastle upon Tyne, 1997.
24. Semenov, A., Yakovlev, A., Pastor, E., Peña, M., Cortadella, J., Lavagno, L.: Partial Order Approach to Synthesis of Speed-Independent Circuits, *Proc. ASYNC'1997*, IEEE Computer Society Press, 1997.
25. Vanbekbergen, P., Catthoor, F., Goossens, G., De Man, H.: Optimized Synthesis of Asynchronous Control Circuits from Graph-Theoretic Specifications, *Proc. ICCAD'1990*, IEEE Computer Society Press, 1990.
26. Yakovlev, A.: Designing Control Logic for Counterflow Pipeline Processor Using Petri nets, *Formal Methods in System Design*, **12**(1), 1998, 39–71.
27. Yakovlev, A., Lavagno, L., Sangiovanni-Vincentelli, A.: A Unified Signal Transition Graph Model for Asynchronous Control Circuit Synthesis, *Formal Methods in System Design*, **9**(3), 1996, 139–188.
28. Zhang, L., Malik, S.: The Quest for Efficient Boolean Satisfiability Solvers, *Proc. CAV'2002* (E. Brinksma, K. Larsen, Eds.), Lecture Notes in Computer Science 2404, Springer-Verlag, 2002.