Detecting State Coding Conflicts in STG Unfoldings Using SAT

Victor Khomenko Maciej Koutny Alex Yakovlev
University of Newcastle upon Tyne, U.K.
{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 that of identifying whether an STG satisfies the Complete State Coding (CSC) requirement, e.g., by using model checking based on the state graph of an STG.

In this paper, we avoid constructing the state 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. The algorithm is derived by adopting the Boolean Satisfiability (SAT) approach. This technique leads not only to huge memory savings when compared to methods based on state graphs, but also to significant speedups.

Keywords: *asynchronous circuits, automated synthesis, complete state coding, CSC, Petri nets, signal transition graphs, STG, SAT, net unfoldings.*

1. Introduction

Signal Transition Graphs (STGs) is a specification language widely used for describing the behaviour of asynchronous control circuits [3, 26]. They are a form 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. One of the commonly used STG-based synthesis tools, PETRIFY [4, 5], performs all of these steps automatically, after first constructing the state graph of the initial STG specification. A vivid example of its use is 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.

In essence, the CSC problem consists in identifying conflicts which occur when semantically different reachable states have the same binary encoding. It is often seen as one which consists of two parts: the detection of coding conflicts, and the elimination of them. The second part may be addressed, for example, by means of changing the causality or ordering constraints (i.e., adding extra places and arcs in the STGs to make implicit timing assumptions explicit), or by introducing 'additional memory' into the system in the form of internal signals. A number of methods for solving the CSC problem are available (see, e.g., [6], for a brief review). Moreover, there are methods using unfoldings rather than state graphs [18]. In this paper, we focus our attention on the first part, namely efficient CSC conflict detection. Besides providing the necessary input for methods resolving CSC conflicts, it can also be seen as a relatively independent problem, as underlined in [1]. There, the first step of the synthesis procedure was to enforce the CSC property by means of the structural insertion of a memory signal for every place in the Petri net underlying the STG. Then a heuristic-driven elimination of those signals was applied iteratively, *while the CSC condition was true.*

Some of CSC conflict detection methods, such as [25], operate directly on the STG level, but they restrict the class of underlying Petri nets to, e.g., marked graphs. Others, such as [4, 5], work in the state graph framework and are general in terms of applicability to the widest possible class of STGs (with bounded underlying Petri Nets). In order to increase efficiency, they often use symbolic (BDD-based) techniques to represent the reachable state space and to capture important relationships (e.g., excitation and quiescent regions, concurrency and conflict relations). While this purely state-based approach is relatively simple and well-studied [5], the issue of computational complexity for highly concurrent STGs is quite serious due to state space explosion. This puts practical bounds on the size of control circuits, which are often restrictive, especially if the STG models are not constructed by a human 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 [23,24]. The idea behind the approach described there was to work with approximate boolean covers obtained for structural elements of the unfolding, namely conditions and events, as opposed to the use of exact boolean covers for markings and excitation regions extracted from the state graph. Although the results were still preliminary, they demonstrated, for some examples, a clear superiority — in terms of memory and time efficiency — of the unfolding-based approach. The main shortcoming of the work described in [23, 24] was that its approximation and refinement strategy was fairly straightforward and could not cope well with the 'don't care' state subsets, i.e., sets of states which would have been unreachable if the exact reachability analysis was applied. [16] has advanced the ideas of slices and cover approximations of [23,24], and presented a theory and algorithms for 'fast' and 'refined' detection of coding conflicts. However, those algorithms have not yet been implemented and proved efficient in experiments, and in their 'refinement' part they still require the construction of the (partial) state space for the subsets of unfolding cuts which evaluate a given boolean cover to true. Bearing this in mind, there is a clear need for further advancement of the unfolding-based methods, both in theory and algorithms, for solving the above mentioned synthesis tasks.

In [11, 13, 14], we proposed a solution for the CSC conflict detection problem. We showed that the notion of a coding conflict can be characterized in terms of a system of integer constraints, and developed an efficient technique for solving such a system. It is also worth pointing out that the method allows one not only to find states which are in coding conflict, but also to derive execution paths leading to them without performing a reachability analysis. That algorithm provided a foundation for the unfolding-based framework for resolution of coding conflicts (step (ii)) described in [18], which used the set of pairs of configurations representing coding conflicts produced by the algorithm. (The method described in this paper is intended to replace that integer programming 'engine' in the framework of [18].)

The integer programming based algorithm in many cases achieved significant speedups, but on some of the benchmarks considered in [11, 13, 14] its performance was still not entirely satisfactory: on several large instances the test did not terminate within the time limit. This is because we used our own specialized integer programming solver, which, though being much more efficient than general-purpose ones due to certain problem-specific heuristics, was not powerful enough to achieve a completely satisfactory running time for large examples. In this paper, we characterize the CSC problem in terms of boolean satisfiability (SAT). Though being more complicated than the integer programming translation, the SAT one allows more dependencies between the variables to be exploited. State-of-the-art SAT solvers are quite efficient, and have been for long employed in the model checking community. In our experiments, we achieved significant speedups using this method.

The technical report [15] shows how the proposed translation can be modified to verify the *USC* and *normalcy* properties, as well as a generalization of the method to STGs with dummy transitions and optimizations possible for certain subclasses of STGs.

2. Basic definitions

A *net* is a triple $N \overset{\text{df}}{=} (P,T,F)$ 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., $M : P \to \mathbb{N} = \{0, 1, 2, \ldots\}$. 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. As usual, ${}^\bullet z \overset{\text{df}}{=} \{y \mid (y, z) \in F\}$ and $z^\bullet \overset{\text{df}}{=} \{y \mid (z, y) \in F\}$ denote, respectively, the *preset* and the *postset* of a node $z \in P \cup T$. We will assume that ${}^\bullet t \neq \emptyset \neq t^\bullet$, for every $t \in T$.

A *net system* is a pair $\Sigma \overset{\text{df}}{=} (N, M_0)$ 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' defined by $M' \overset{\text{df}}{=} M - {}^\bullet t + t^\bullet$, 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 *Signal Transition Graph (STG)* is a triple $\Gamma \overset{\text{df}}{=} (\Sigma, Z, \lambda)$ such that $\Sigma = (N, M_0)$ is a net system, Z is a finite set of signals, generating a finite alphabet $Z^\pm \overset{\text{df}}{=} Z \times \{+, -\}$ of *signal transition labels*, and $\lambda : T \to Z^\pm$ is a labelling function. The signal transition labels are of the form z^+ or z^-, and denote a transition of a signal $z \in Z$ from 0 to 1 (rising edge), or from 1 to 0 (falling edge), respectively. We will also 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.

In addition to the drawing conventions for Petri nets, we use the following one for STGs. When an arc in a figure connects two transitions, it is assumed that there is a place 'in the middle' of the arc. Moreover, an arc with a token on it is interpreted similarly, but the place contains a token.

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

Γ is *consistent*if, for every reachable marking M, all firing sequences σ from M_0 to M have the same *encoding vector* $Code(M)$ equal to $v^0 + v^\sigma$, and this vector is binary, i.e., $Code(M) \in \{0, 1\}^{|Z|}$. 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.[1] 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^0_z and p^1_z, tracing the value of z as follows. Each z^+–labelled transition has p^0_z in its preset and p^1_z in its postset, and each z^-–labelled transition has p^1_z in its preset and p^0_z in its postset. Exactly one of these two places is marked at the initial state, accordingly to the initial value of signal z.[2] One can show that at any reachable state of an STG augmented with such places, p^0_z (resp. p^1_z) is marked iff the value of z is 0 (resp. 1). Thus, if a transition labelled by z^+ (resp. z^-) is enabled then the value of z is 0 (resp. 1), which in turn guarantees the consistency. Such a transformation can be done completely automatically. For a consistent STG, it does not restrict the behaviour and yields an STG with isomorphic *state graph*, defined below; for a non-consistent STG, this transformation restricts the behaviour and may lead to (new) deadlocks. In what follows, we assume such tracing places in the STG, and denote $P^0_Z \overset{\text{df}}{=} \{p^0_z \mid z \in Z\}$, $P^1_Z \overset{\text{df}}{=} \{p^1_z \mid z \in Z\}$, and $P_Z \overset{\text{df}}{=} P^0_Z \cup P^1_Z$.

The *state graph* of an STG Γ is a tuple $SG_\Gamma \overset{\text{df}}{=} (S, A, M_0, Code)$ such that: $S \overset{\text{df}}{=} [M_0\rangle$ is the set of *states*; $A \overset{\text{df}}{=} \{M \xrightarrow{t} M' \mid M \in [M_0\rangle \wedge M[t\rangle M'\}$ is the set of *state*

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

[2]In practice, this transformation is performed on the prefix rather than the original STG, and hence the initial values of all the signals can easily be computed.

transitions; M_0 is the *initial state*; and $Code : S \to \{0,1\}^{|Z|}$ is the *state assignment* function, as defined above for markings.

It is often the case that Z is 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 logical 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: $Nxt_z(M) \overset{\text{df}}{=} 0$ if $Code_z(M) = 0$ and no z^+–labelled transition is enabled at M, or $Code_z(M) = 1$ and a z^-–labelled transition is enabled at M; and $Nxt_z(M) \overset{\text{df}}{=} 1$ if $Code_z(M) = 1$ and no z^-–labelled transition is enabled at M, or $Code_z(M) = 0$ 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., $Nxt_z(M)$ should be a function of $Code(M)$: $Nxt_z(M) = F_z(Code(M))$ (F_z will eventually be implemented as a logic gate). To capture this, let $Out(M) \overset{\text{df}}{=} \{z \in Z_O \mid \exists t \in T : M[t\rangle \wedge \lambda(t) = z^{\pm}\}$ be the set of enabled output signals, for every reachable state $M \in S$. We also define $Out_z(M)$ to be 1 if $z \in Out(M)$ and 0 otherwise. Two states of SG_Γ are in *CSC conflict* if they have the same encoding but different sets of enabled output signals. Γ satisfies the *Complete State Coding (CSC)* property if no two states of SG_Γ are in CSC conflict.

An example of an STG for a data read operation in a simple VME bus controller (a standard STG benchmark, see, e.g., [5]) is shown in Figure 1(a). Part (b) of this figure illustrates a CSC conflict between two different states, M' and M'', that have the same code, 10110, but $Out(M') = \{d\} \neq Out(M'') = \{lds\}$. This means that, e.g., the value of $F_{lds}(1,0,1,1,0)$ is ill-defined (it should be 1 according to the state M' and 0 according to the state M''), and thus lds is not implementable as a logic gate. To cope with this, an additional signal helping to resolve this CSC conflict should be added to the STG.

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 ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$ 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 $ON \overset{\text{df}}{=} (B,E,G)$ where B is the set of *conditions* (places) and E is the set of *events* (transitions). 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 $y' \preceq y$.

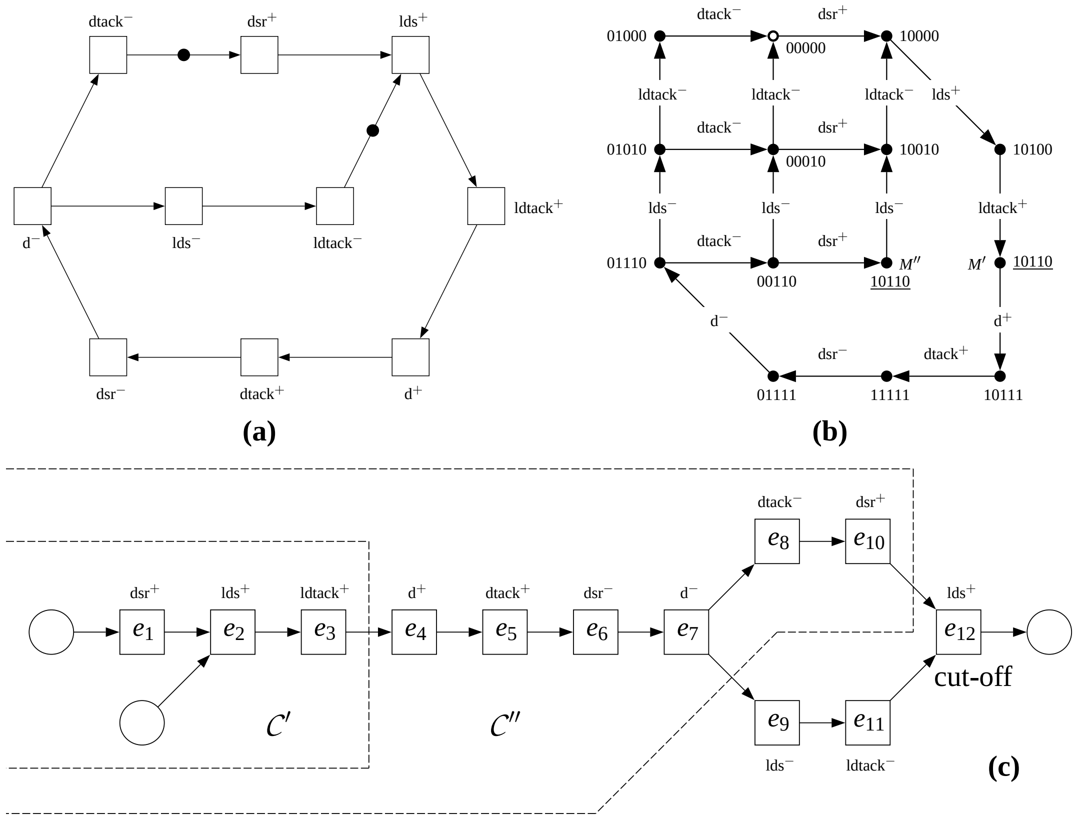
A *homomorphism* from an occurrence net ON to a net system Σ is a mapping $h : B \cup E \to P \cup T$ such that: $h(B) \subseteq P$ and $h(E) \subseteq T$; for all $e \in E$, the restriction of h to ${}^\bullet e$ is a bijection between ${}^\bullet e$ and ${}^\bullet h(e)$; the restriction of h to $e^\bullet$ is a bijection between $e^\bullet$ and $h(e)^\bullet$; the restriction of h to $Min(ON)$ is a bijection between $Min(ON)$ and M_0; and for all $e, f \in E$, if ${}^\bullet e = {}^\bullet f$ and $h(e) = h(f)$ then $e = f$.

A *branching process* of Σ [7] is a quadruple $\pi \overset{\text{df}}{=} (B,E,G,h)$ such that (B,E,G) is an occurrence net and h is a homomorphism from ON 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.

A *configuration* of an occurrence net ON is a set of events $\mathcal{C}$ such that for all $e, f \in \mathcal{C}$, $\neg(e\#f)$ and, for every $e \in \mathcal{C}$, $f \prec e$ implies $f \in \mathcal{C}$. 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 $\mathcal{C}$ be a finite configuration of a branching process π. Then $Cut(\mathcal{C}) \overset{\text{df}}{=} (Min(ON) \cup \mathcal{C}^\bullet) \setminus {}^\bullet\mathcal{C}$ is a cut; moreover, the multiset of places $h(Cut(\mathcal{C}))$ is a reachable marking of Σ, denoted $Mark(\mathcal{C})$. A marking M of Σ is *represented* in π if the latter contains a finite configuration $\mathcal{C}$ such that $M = Mark(\mathcal{C})$. Every marking represented in π is reachable, and every reachable marking is represented in the unfolding of Σ [11,12,19,20,23].

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



$$\begin{aligned}
\mathsf{conf}' &= 111000000000 & & & \mathsf{conf}'' &= 111111110100\\
\mathsf{en}' &= 000100000000 & \mathsf{val} &= 10110 & \mathsf{en}'' &= 000000001000\\
\mathsf{out}' &= 00001 & & & \mathsf{out}'' &= 00100
\end{aligned}$$

Figure 1. An STG modelling a simplified VME bus controller (a); its state graph with a CSC conflict between two states (b); and its unfolding prefix with two configurations corresponding to this CSC conflict (c). The order of signals in the binary codes is: $dsr, dtack, lds, ldtack, d$**.**

a finite configuration $\mathcal{C}$ of π such that $\mathcal{C} \cap E_{cut} = \emptyset$ and $M = Mark(\mathcal{C})$, and for every transition t enabled by M, there is an event $e \notin \mathcal{C}$ in π such that $h(e) = t$ and $\mathcal{C} \cup \{e\}$ is a configuration (e may be a cut-off event).

Although, in general, unfoldings are infinite, for a bounded net system Σ one can construct a finite complete prefix Unf_Σ of the unfolding of Σ, by choosing an appropriate set E_{cut} of cut-off events, beyond which the unfolding is not generated [11, 12, 19, 20, 23].

A *branching process* of an STG $\Gamma = (\Sigma, Z, \lambda)$ is a branching process of Σ augmented with an additional labelling of its events, $\lambda \circ h : E \to Z^{\pm}$.

We also extend the functions *Code* and Out_z to finite configurations of branching processes of Γ in the following way: $Code(\mathcal{C}) \overset{\text{df}}{=} Code(Mark(\mathcal{C}))$ and $Out_z(\mathcal{C}) \overset{\text{df}}{=} Out_z(Mark(\mathcal{C}))$.

The *boolean satisfiability problem (SAT)* 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. This expression is often assumed to be given in *conjunctive normal form (CNF)* $\bigwedge_{i=1}^{n} \bigvee_{l \in L_i} l$, i.e., it should be represented as a conjunction of *clauses*, which are disjunctions of *literals*. Each literal is either a variable or its negation. 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 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 following *propagation rule*: "If all but one variables occurring in some clause have been assigned a value such that the corresponding literals in this clause have the value 0 then in order to satisfy the clause the remaining variable should be assigned a value such that the corresponding literal would 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 of some clause are assigned the value 0 then the built partial assignment cannot be a part of any satisfying assignment, and the solver should backtrack.

3. State coding conflict detection using SAT

Given a finite complete prefix $\pi = (B,E,G,h)$ of the unfolding of an STG, a CSC violation can be represented as a pair of configurations, $\mathcal{C}'$ and $\mathcal{C}''$, whose final states are in CSC conflict, as shown Figure 1(c). The following boolean variables will be used in our translation of this property into a SAT problem (Figure 1 shows the values of these variables for the depicted CSC conflict):

- For $e \in E \setminus E_{cut}$, where E_{cut} are the cut-off events of the prefix, we have variables conf'_e and conf''_e, tracing whether $e \in \mathcal{C}'$ and $e \in \mathcal{C}''$ respectively.

- For $z \in Z$, we have a variable val_z tracing the value of z. Since the values of all the signals must match at the final states of $\mathcal{C}'$ and $\mathcal{C}''$, we use the same set of variables for both configurations.

- For $b \in B \setminus E^{\bullet}_{cut}$ such that $h(b) \in P^1_z$, we have variables cut'_b and cut''_b, tracing whether $b \in Cut(\mathcal{C}')$ and $b \in Cut(\mathcal{C}'')$ respectively.

- For $z \in Z_O$, we have variables out'_z and out''_z, tracing whether $z \in Out(\mathcal{C}')$ and $z \in Out(\mathcal{C}'')$ respectively.

- For $e \in E$, we have variables en'_e and en''_e, tracing whether e is 'enabled' by $\mathcal{C}'$ and $\mathcal{C}''$ respectively.

Our aim is to build a boolean formula $\mathcal{CSC}$ such that: (i) $\mathcal{CSC}$ is satisfiable iff there is a CSC conflict; and (ii) for every satisfying assignment, the two sets of non-cut-off events of the prefix, $\mathcal{C}' \overset{\text{df}}{=} \{e \mid \mathsf{conf}'_e = 1\}$ and $\mathcal{C}'' \overset{\text{df}}{=} \{e \mid \mathsf{conf}''_e = 1\}$, constitute a pair of configurations representing a CSC violation. $\mathcal{CSC}$ will be the conjunction of constraints described below.

3.1. Configuration constraints

The role of configuration constraints, $\mathcal{CONF}'$ and $\mathcal{CONF}''$, is to ensure that $\mathcal{C}'$ and $\mathcal{C}''$ are both legal configurations of the prefix. $\mathcal{CONF}'$ 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

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

The former formula ensures that $\mathcal{C}'$ is downward closed w.r.t. $\preceq$, and the latter one ensures that $\mathcal{C}'$ contains no structural conflicts. $\mathcal{CONF}''$ is defined similarly.

3.2. Encoding constraints

The role of encoding constraints, $\mathcal{CODE}'$ and $\mathcal{CODE}''$, is to ensure that the signal codes of the final markings of configurations $\mathcal{C}'$ and $\mathcal{C}''$ are equal. To build a formula establishing the value val_z of each signal $z \in Z$ at the final state of $\mathcal{C}'$, we observe that $\mathsf{val}_z = 1$ iff $p^1_z \in Mark(\mathcal{C}')$, i.e., iff $b \in Cut(\mathcal{C}')$ for some p^1_z–labelled condition b (note that the places in P_Z cannot contain more than one token). The latter can be captured by the constraint:

$$\bigwedge_{z \in Z} (\mathsf{val}_z \iff \bigvee_{b \in B_z} \mathsf{cut}'_b) \ ,$$

where $B_z \overset{\text{df}}{=} \{B \setminus E^{\bullet}_{cut} \mid h(b) = p^1_z\}$. We then define $\mathcal{CODE}'$ as the conjunction of the last formula and

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

which ensures that $b \in Cut(\mathcal{C}')$ 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 $\mathcal{C}'$ is a configuration and $\mathcal{CODE}'$ is satisfied then the value of the signal z at the final state of $\mathcal{C}'$ is given by val_z. *Code*$''$ is defined similarly.

Using the same variables val_z in both $\mathcal{CODE}'$ and $\mathcal{CODE}''$, ensures that the encodings of the final states of $\mathcal{C}'$ and $\mathcal{C}''$ are the same, if both constraints are satisfied.

3.3. Separating constraint

The role of the separating constraint $\mathcal{SEP}$ is to ensure that the sets of output signals enabled at the final markings of configurations $\mathcal{C}'$ and $\mathcal{C}''$ are different. We observe that $z \in Z_O$ is enabled by the final state of $\mathcal{C}'$ iff there is a $z^\pm$–labelled event $e \notin \mathcal{C}'$ 'enabled' by $\mathcal{C}'$, i.e., such that $\mathcal{C}' \cup \{e\}$ is a configuration (note that e can be a cut-off event). We then define $\mathcal{OUT}'$ as the conjunction of

$$\bigwedge_{z\in Z_O} (\mathsf{out}'_z \iff \bigvee_{e\in E_z} \mathsf{en}'_e)$$

and

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

where $E_z \overset{\text{df}}{=} \{e \in E \mid \lambda(h(e)) = z^\pm\}$. One can see that $\mathcal{OUT}'$ is satisfied iff the variables en'_e show for each event whether it is enabled by $\mathcal{C}'$, and the values of the variables out'_z correspond to the values of the output signals enabled by the final state of $\mathcal{C}'$. $\mathcal{OUT}''$ is defined similarly, and so we can express the non-equality of enabled outputs as the constraint $\mathcal{SEP}$, defined by:

$$\bigvee_{z\in Z_O} (\mathsf{out}'_z \neq \mathsf{out}''_z)\ .$$

Intuitively, the formulae $\mathcal{OUT}'$ and $\mathcal{OUT}''$ determine the sets of output signals enabled by the final states of $\mathcal{C}'$ and $\mathcal{C}''$, and $\mathcal{SEP}$ requires these sets to be distinct.

3.4. Translation to SAT

Finally, the problem at hand can be formulated as a SAT problem for the formula

$$\mathcal{CSC} \overset{\text{df}}{=} \mathcal{CONF}' \wedge \mathcal{CONF}'' \wedge \mathcal{CODE}' \wedge \mathcal{CODE}'' \wedge \mathcal{OUT}' \wedge \mathcal{OUT}'' \wedge \mathcal{SEP}\ .$$

It can be easily translated into the conjunctive normal form [15] and fed to a SAT solver.

4. Experimental results

We implemented our method using the zCHAFF SAT solver [21] which was available from the Internet. For testing the performance of the proposed approach we used the same benchmarks as in [11,13,14]. The STGs with names containing the occurrence of 'CSC' satisfy the CSC property, the others exhibit CSC conflicts. All the experiments were conducted on a PC with *Pentium*^{TM} III/500MHz processor and 384M RAM.

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

- LAZYRING and RING — Asynchronous Token Ring Adapters described in [2,17].
- DUP4PH, DUP4PHCSC, DUP4PHMTR, DUP4PHMTRCSC, DUPMTRMOD, DUPMTRMODUTG, and DUPMTRMODCSC — control circuits for the Power-Efficient Duplex Communication System described in [9].
- CFSYMCSCA, CFSYMCSCB, CFSYMCSCC, CFSYMCSCD, CFASYMCSCA, and CFASYMCSCB — control circuits for the Counterflow Pipeline Processor described in [27].

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, PPWK(m,n) and PPARB(m,n), contain scalable examples of STGs modelling m pipelines weakly synchronized without arbitration (in PPWK(m,n)) and with arbitration (in PPARB(m,n)). These two groups of benchmarks allowed us to test the algorithm on pairs of almost identical specifications, such that one element of each pair contains a coding conflict while the other satisfies the CSC property (see [11,15] for more details). The results for these two groups are summarized in the last two parts of Table 1.

The meaning of the columns is as follows (from left to right): the name of the problem; the number of places, transitions, and signals in the original STG;

Problem	Net			Unfolding			Formula			Time, [s]		
	$\|S\|$	$\|T\|$	$\|Z\|$	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	*Var*	*Cl*	*Lit*	Pfy	Clp	Sat
Real-Life STGs												
LazyRing	35	32	11	87	66	5	295	765	1790	2.67	0.01	0.01
Ring	147	127	28	763	498	59	2075	6233	14793	1762	0.42	0.41
Dup4ph	133	123	26	144	123	11	576	1469	3455	45.01	<0.01	0.06
Dup4phCsc	135	123	26	146	123	11	572	1477	3479	43.50	0.03	0.08
Dup4phMtr	109	96	22	117	96	8	444	1205	2840	30.37	<0.01	0.03
Dup4phMtrCsc	114	105	26	122	105	8	510	1325	3120	31.85	0.01	0.05
DupMtrMod	129	100	21	199	132	10	580	1651	3869	383	<0.01	0.05
DupMtrModUtg	116	165	21	344	218	65	862	3097	7361	721	<0.01	0.05
DupMtrModCsc	152	115	27	228	149	13	678	1935	4545	387	0.09	0.50
CfSymCscA	85	60	22	1341	720	56	2920	10247	24502	494	91.78	16.86
CfSymCscB	55	32	16	160	71	6	420	1357	3256	19.28	0.05	0.03
CfSymCscC	59	36	18	286	137	10	758	2551	6170	42.15	2.16	0.72
CfSymCscD	45	28	14	120	54	6	332	1031	2452	10.56	0.01	0.01
CfAsymCscA	128	112	34	1808	1234	62	5312	16687	39964	4357	2514	179
CfAsymCscB	128	112	32	1816	1238	62	5108	15977	38226	8490	2687	129
Marked Graphs												
PpWk(2,3)	23	14	7	41	23	1	146	407	967	0.71	<0.01	<0.01
PpWk(2,6)	47	26	13	119	62	1	368	1115	2677	15.82	0.01	0.03
PpWk(2,9)	71	38	19	233	119	1	680	2147	5179	128	0.28	0.08
PpWk(2,12)	95	50	25	383	194	1	1082	3503	8473	1301	3.66	0.34
PpWkCsc(2,3)	24	14	7	38	20	1	130	359	851	0.56	<0.01	0.01
PpWkCsc(2,6)	48	26	13	110	56	1	338	1015	2433	13.56	0.16	0.16
PpWkCsc(2,9)	72	38	19	218	110	1	634	1991	4799	124	12.98	2.13
PpWkCsc(2,12)	96	50	25	362	182	1	1022	3295	7965	5389	1106	6.92
PpWk(3,3)	34	20	10	63	35	1	220	621	1480	3.29	<0.01	<0.01
PpWk(3,6)	70	38	19	183	95	1	560	1709	4109	277	0.56	<0.01
PpWk(3,9)	106	56	28	357	182	1	1036	3285	7930	5555	51.36	0.23
PpWk(3,12)	142	74	37	585	296	1	1646	5345	12935	*time*	4542	1.83
PpWkCsc(3,3)	36	20	10	57	29	1	188	525	1248	2.40	0.01	0.01
PpWkCsc(3,6)	72	38	19	165	83	1	500	1509	3621	122	11.81	0.25
PpWkCsc(3,9)	108	56	28	327	164	1	944	2973	7170	18568	8990	0.95
PpWkCsc(3,12)	144	74	37	543	272	1	1526	4929	11919	*mem*	*time*	19.23
STGs with Arbitration												
PpArb(2,3)	38	24	11	94	52	2	308	973	2347	4.09	<0.01	0.02
PpArb(2,6)	62	36	17	202	106	2	608	1993	4825	45.42	<0.01	0.03
PpArb(2,9)	86	48	23	346	178	2	992	3325	8071	912	0.02	0.06
PpArb(2,12)	110	60	29	526	268	2	1472	4993	12133	5737	0.03	0.19
PpArbCsc(2,3)	40	24	11	96	52	2	308	973	2343	3.53	0.03	0.05
PpArbCsc(2,6)	64	36	17	204	106	2	608	1993	4821	53.39	2.00	0.59
PpArbCsc(2,9)	88	48	23	348	178	2	992	3325	8067	458	174	2.34
PpArbCsc(2,12)	112	60	29	528	268	2	1472	4993	12129	*mem*	15158	12.73
PpArb(3,3)	56	36	16	161	90	3	520	1821	4442	25.25	<0.01	0.05
PpArb(3,6)	92	54	25	341	180	3	1018	3627	8855	1158	<0.01	0.27
PpArb(3,9)	128	72	34	575	297	3	1636	5889	14396	8410	0.02	0.67
PpArb(3,12)	164	90	43	863	441	3	2404	8667	21185	*mem*	0.05	5.73
PpArbCsc(3,3)	59	36	16	164	90	3	520	1821	4436	18.31	0.34	0.16
PpArbCsc(3,6)	95	54	25	344	180	3	1018	3627	8849	510	262	2.22
PpArbCsc(3,9)	131	72	34	578	297	3	1636	5889	14390	11081	*time*	9.53
PpArbCsc(3,12)	167	90	43	866	441	3	2404	8667	21179	*mem*	*time*	49.81

Table 1. Experimental results.

the number of conditions, events and cut-off events in the complete prefix; the number of variables, clauses and literals in the generated formula; the time spent by a special version of the Petrify tool, which did not attempt to resolve the coding conflicts it had identified; the time spent by the integer programming algorithm proposed in [11, 13, 14]; and the time spent by the method proposed in this paper. We use 'mem' if there was 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. This can be explained by the fact that STGs usually contain a lot of concurrency but relatively few choices, and thus the prefixes are often not much bigger then the STGs themselves. As a result, unfolding-based methods have clear advantage (in contrast, PETRIFY was repeatedly swapping pages to the disk for some of the examples due to the need to build the whole state spaces of the STGs).

Although 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 the other approaches. In some cases, it was faster by several orders of magnitude. The time spent on all of these benchmarks was quite satisfactory — it took less than 3 minutes to solve the hardest one. Overall, the proposed approach was the best, 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 propagation rule by the SAT solver.

5. 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 then 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 [18] they form a framework for detection and solving encoding conflicts, which does not require generating the STG's state space. In our future work, we plan to develop a complete design flow for asynchronous circuits based on STG unfolding prefixes.

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., [10, 11, 13, 14]). 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 then be translated into some canonical problem, e.g., an integer programming one [11, 13, 14], a problem of finding a stable model of a logic program [10], or a boolean satisfiability problem as in this paper. Then an appropriate solver can be used for efficiently solving it.

Acknowledgements

We would like to thank Jordi Cortadella for compiling a special version of the PETRIFY tool used in our experiments. The first author was supported by an ORS Awards Scheme grant ORS/C20/4 and by an EPSRC grant GR/M99293. The other two authors were supported by an EPSRC grant GR/M94366 (MOVIE).

References

[1] J. Carmona, J. Cortadella, and E. Pastor: A Structural Encoding Technique for the Synthesis of Asynchronous Circuits. Proc. of *ICACSD'01*, IEEE Comp. Soc. Press (2001) 157–166.

[2] C. Carrion and A. Yakovlev: Design and Evaluation of Two Asynchronous Token Ring Adapters. Tech. Rep. CS-TR-562, Univ. of Newcastle upon Tyne (1996).

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

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

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

[6] J. Cortadella, A. Kondratyev, M. Kishinevsky, L. Lavagno, and A. Yakovlev: Complete State Encoding Based on Theory of Regions. Proc. of *ASYNC'1996*, IEEE Comp. Soc. Press (1996) 36–47.

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

[8] J. Esparza: Decidability and Complexity of Petri Net Problems — an Introduction. In: *Lectures on Petri Nets I: Basic Models*, W. Reisig and G. Rozenberg (Eds.). LNCS 1491 (1998) 374–428.

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

[10] K. Heljanko: Using Logic Programs with Stable Model Semantics to Solve Deadlock and Reachability Problems for 1-Safe Petri Nets. *Fundamentae Informaticae* 37(3) (1999) 247–268.

[11] V. Khomenko: *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD Thesis, University of Newcastle upon Tyne (2003).

[12] V. Khomenko, M. Koutny, and V. Vogler: Canonical Prefixes of Petri Net Unfoldings. Proc. of *CAV'2002*, LNCS 2404 (2002) 582–595.

[13] V. Khomenko, M. Koutny, and A. Yakovlev: Detecting State Coding Conflicts in STGs Using Integer Programming. Tech. Rep. CS-TR-736, Univ. of Newcastle upon Tyne (2001).

[14] V. Khomenko, M. Koutny, and A. Yakovlev: Detecting State Coding Conflicts in STGs Using Integer Programming. Proc. of *DATE'2002*, IEEE Comp. Soc. Press (2002) 338–345.

[15] V. Khomenko, M. Koutny, and A. Yakovlev: Detecting State Coding Conflicts in STG Unfoldings Using SAT. Tech. Rep. CS-TR-778, Univ. of Newcastle upon Tyne (2002).

[16] A. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, A. Taubin, and A. Yakovlev: Identifying State Coding Conflicts in Asynchronous System Specifications Using Petri Net Unfoldings. Proc. of *ICACSD'98*, IEEE Comp. Soc. Press (1998) 152–163.

[17] K. S. Low and A. Yakovlev: Token Ring Arbiters: an Exercise in Asynchronous Logic Design with Petri Nets. Tech. Rep. CS-TR-537, Univ. of Newcastle upon Tyne (1995).

[18] A. Madalinski, A. Bystrov, V. Khomenko, and A. Yakovlev: Visualization and Resolution of Coding Conflicts in Asynchronous Circuit Design. Proc. of *DATE'2003*, IEEE Comp. Soc. Press (2003) 926–931.

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

[20] K. L. McMillan: *Symbolic Model Checking: an approach to the state explosion problem*. PhD thesis, CMU-CS-92-131 (1992).

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

[22] E. Pastor, J. Cortadella, and O. Roig: Symbolic Analysis of Bounded Petri Nets. *IEEE Transactions on Computers* 50(5) (2001) 432–448.

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

[24] A. Semenov, A. Yakovlev, E. Pastor, M. Peña, J. Cortadella, and L. Lavagno: Partial Order Approach to Synthesis of Speed-Independent Circuits. Proc. of *ASYNC'1997*, IEEE Comp. Soc. Press (1997) 254–265.

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

[26] 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.

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

[28] L. Zhang and S. Malik: The Quest for Efficient Boolean Satisfiability Solvers. Proc. of *CAV'2002*, LNCS 2404 (2002) 582–595.