# Detecting State Coding Conflicts in STGs
# Using Integer Programming

Victor Khomenko, Maciej Koutny, and Alex Yakovlev
Department of Computing Science, University of Newcastle
Newcastle upon Tyne NE1 7RU, U.K.
{Victor.Khomenko, Maciej.Koutny, Alex.Yakovlev}@ncl.ac.uk

## Abstract

*The paper presents a new method for checking Unique and Complete State Coding, the crucial conditions in the synthesis of asynchronous control circuits from Signal Transition Graphs (STGs). The method detects state coding conflicts in an STG using its partial order semantics (unfolding prefix) and an integer programming technique. This leads to huge memory savings compared to methods based on reachability graphs, and also to significant speedups in many cases. In addition, the method produces execution paths leading to an encoding conflict. Finally, the approach is extended to checking the normalcy property of STGs, which is a necessary condition for their implementability using gates whose characteristic functions are monotonic.*

## 1. Introduction

Signal Transition Graphs (STGs) are widely used for synthesis of asynchronous control circuits. STGs are interpreted Petri nets in which transitions are labelled with the rising and falling edges of circuit signals ([2]). Synthesis based on STGs involves: (a) checking the necessary and sufficient conditions for STG's implementability as a logic circuit; (b) modifying, if necessary, the initial STG to make it implementable; and (c) finding appropriate boolean covers for the next-state functions of output and internal signals. One of the commonly used tools, `Petrify` ([3]), performs all of these steps automatically, after first constructing the reachability graph of the initial STG specification.

To gain efficiency, `Petrify` uses symbolic (BDD-based) techniques to represent the STG reachable state space. While the state-based approach is convenient for finding good synthesis solutions, the combinatorial explosion of the state space is a serious issue for highly concurrent STGs (e.g. generated from high-level hardware descriptions). This paper investigates a way to alleviate state space explosion by using causal partial order semantics of STGs, in the form of a Petri net unfolding prefix ([15]).

We apply unfoldings to the implementability analysis in step (a), viz. checking the Complete State Coding (CSC) and the Unique State Coding (USC) conditions ([2]), which requires detecting coding conflicts between markings. A number of methods for detecting and resolving such conflicts exist to date (see, e.g., [4] for a brief review). Most of them work in the state graph framework and are applicable to the widest possible class of STGs (with bounded underlying Petri Nets). Some, such as [17], operate directly on the STG level but are restricted to, e.g., marked graphs.

STG unfoldings were first applied to the detection of state conflicts in [10], where the ideas of slices and cover approximations of [15] led to theory and algorithms for 'fast' and 'refined' detection of coding conflicts. However, those algorithms have not been implemented and proved efficient in experiments, and in their 'refinement' part they still require the construction of the (partial) state space

In this paper, another kind of unfolding-based approach is developed, in which we characterise state coding conflict in terms of a system of integer constraints. Moreover, the causality and conflicts between events involved in an unfolding impose certain relationship between the corresponding variables in the system of constraints, which can be used to speed up the algorithm. Our new approach to state coding conflict detection is in some sense opposite to that of the state graph based one, and exploits only the characteristics of the unfolding structure itself. Unlike [10], our method is not concerned with boolean covers for parts of the unfolding. The initial motivation for applying this technique to the problem of CSC (and USC) comes from its remarkable success in speeding up deadlock detection ([8]) and solving some other model-checking problems. Our experiments demonstrate that, in addition to huge memory savings, the proposed algorithm can often achieve significant speedups. It is also worth pointing out that the method allows one not only to find conflicting reachable states, but also execution paths leading to them without performing a reachability analysis. We also apply this technique to checking the normalcy property of STGs, which is a necessary condition for their implementability using gates described by monotonic functions (e.g. standard NOT, NAND, NOR, AOI, OAI gates). Details of theory, including proofs, underlying our method can be found in the full version [9].

## 2. Basic definitions

In this section, we first present basic definitions concerning Petri nets and STGs, and then recall (see also [6]) notions related to net unfoldings.

### 2.1. Petri nets and STGs

A *net* is a triple $N \overset{\text{df}}{=} (S, T, F)$ such that $S$ and $T$ are disjoint sets of respectively *places* (circles) and *transitions* (boxes), collectively known as *nodes*, and $F \subseteq (S \times T) \cup (T \times S)$ is a *flow relation*. We denote ${}^\bullet z \overset{\text{df}}{=} \{y \mid (y, z) \in F\}$ and $z^\bullet \overset{\text{df}}{=} \{y \mid (z, y) \in F\}$, for all $z \in S \cup T$, and assume that ${}^\bullet t \neq \emptyset \neq t^\bullet$, for every $t \in T$. A *marking* of $N$ is a multiset $M$ of places, i.e., $M : S \to \mathbb{N} = \{0, 1, 2, \ldots\}$.

A *net system* is a pair $\Sigma \overset{\text{df}}{=} (N, M_0)$ comprising a finite net $N = (S, T, F)$ 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 $\Sigma$ is the smallest (w.r.t. set inclusion) 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 $\Sigma$ is *bounded* if there is $k \in \mathbb{N}$ such that, for every reachable marking $M$, $M(S) \subseteq \{0, \ldots, k\}$; in particular, it is safe when $k = 1$.

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, which generate a finite alphabet $Z^{\pm} \overset{\text{df}}{=} Z \times \{+, -\}$ of *signal transition labels*, and $\lambda : T \to Z^{\pm} \cup \{\tau\}$ is a labelling function, where $\tau$ is a label indicating an silent (dummy) transition. The signal transition labels are of the form $z+$ or $z-$, and denote the transitions of signals $z \in Z$ 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 interested in its direction.

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

$\Gamma$ is *consistent*[1] if, for every reachable marking $M$, all firing sequences $\sigma$ from $M_0$ to $M$ have the same *encoding vector* $Code(M) \overset{\text{df}}{=} v^0 + v^\sigma$, 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 properties: (i) the first occurrence of $z$ in the labelling of any firing sequence of $\Gamma$ starting from $M_0$ has the same sign (either rising of falling); and (ii) the rising and falling labels $z$ alternate in any firing sequence of $\Gamma$. All STGs considered in the sequel are assumed to be consistent.

The *state graph* of $\Gamma$ is a tuple $SG_\Gamma \overset{\text{df}}{=} (S, A, s_0, Code)$ 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 *transitions*; $s_0 \overset{\text{df}}{=} M_0$ is the *initial state*; and $Code : S \to \{0, 1\}^{|Z|}$ is the *state assignment* function, as defined above for markings.

Two distinct states of $SG_\Gamma$ are in *USC conflict* if they are assigned the same code. $\Gamma$ satisfies the *Unique State Coding (USC)* property if no two states of $SG_\Gamma$ are in USC conflict.

Signals in $Z$ are often 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 in the circuit. Logic synthesis derives boolean equations for the output signals, which requires the conditions for enabling output signal transitions in the state graph of the STG to be defined without ambiguity. 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 $M$. Two distinct states of $SG_\Gamma$ are in *CSC conflict* if they have the same code but the sets of enabled output signals are different. $\Gamma$ satisfies the *Complete State Coding (CSC)* property if no two states of $SG_\Gamma$ are in CSC conflict.

In this paper, we will assume that the considered STGs contain no $\tau$-labelled transitions. The case of an STG with dummy transitions is elaborated in the full version [9].

An example of an STG for a data read operation in a simple VME bus controller (a standard STG benchmark) is shown in Fig. 1(a). Part (b) of this figure illustrates CSC conflict between two different markings, $M'$ and $M''$, that have the same code, 10110, but $Out(M') = \{lds\} \neq Out(M'') = \{d\}$.

[1] This is a somewhat simplified notion of consistency; see [15] for a more elaborated one.

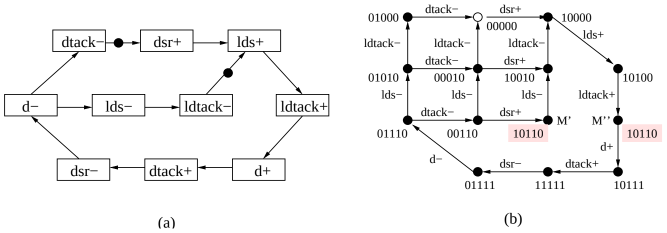

**Figure 1. STG model of a simplified VME bus controller (a) and its state graph with a CSC conflict between two states (b). The order of signals in the binary codes is: $dsr, dtack, lds, ldtack, d$. Places with one incoming and one outgoing arc are not shown.**

### 2.2. Marking equation

Let $\Sigma = (N, M_0)$ be a net system, and $S = \{s_1, \ldots, s_m\}$ and $T = \{t_1, \ldots, t_n\}$ be sets of its places and transitions, respectively. We will often identify a marking $M$ of $\Sigma$ with a vector $(\mu_1, \ldots, \mu_m)$ such that $M(s_i) = \mu_i$, for all $i \leq m$. The *incidence matrix* of $\Sigma$ is an $m \times n$ matrix $I = (I_{ij})$ such that, for all $i \leq m$ and $j \leq n$,

$$I_{ij} \overset{\text{df}}{=} \begin{cases} \phantom{-}1 & \text{if } s_i \in t_j^\bullet \setminus {}^\bullet t_j \\ -1 & \text{if } s_i \in {}^\bullet t_j \setminus t_j^\bullet \\ \phantom{-}0 & \text{otherwise .} \end{cases}$$

The *Parikh vector* of a finite sequence of transitions $\sigma$ is a vector $x_\sigma = (x_1, \ldots, x_n)$ such that $x_i$ is the number of the occurrences of $t_i$ within $\sigma$, for every $i \leq n$. One can show that if $\sigma$ is an execution sequence such that $M_0[\sigma\rangle M$ then $M = M_0 + I \cdot x_\sigma$. This provides a motivation for investigating the feasibility (or solvability) of the following system of equations:

$$\begin{cases} M = M_0 + I \cdot x \\ M \in \mathbb{N}^m \text{ and } x \in \mathbb{N}^n \text{ .} \end{cases} \tag{1}$$

If we fix the marking $M$, then the feasibility of the above system is a necessary (but, in general, not sufficient) condition for $M$ to be reachable from $M_0$.

A vector $x \in \mathbb{N}^n$ is $\Sigma$-*compatible* if it is the Parikh vector of some execution sequence of $\Sigma$. Each compatible vector is a solution of the marking equation for some reachable marking $M$, but, in general, (1) can have solutions which do not correspond to any execution sequence of $\Sigma$. However, for the class of acyclic nets (in particular, net unfoldings), this equation provides an *exact* characterisation of the set of reachable markings ([14]) — the fact which is crucial for the approach proposed in this paper.

### 2.3. Branching processes and configurations

Two nodes of a net $N = (S, T, F)$, $y$ and $y'$, are *in 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 *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 $\Sigma$ is a mapping $h : B \cup E \rightarrow S \cup T$ such that: $h(B) \subseteq S$ 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 $\Sigma$ ([5]) 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 $\Sigma$. A branching process $\pi' = (B', E', G', h')$ of $\Sigma$ 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 $\Sigma$ there exists a unique (up to isomorphism) maximal (w.r.t. $\sqsubseteq$) branching process, called the *unfolding* of $\Sigma$.

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 $f \in \mathcal{C}$. The configuration $[e] \overset{\text{df}}{=} \{f \mid f \preceq e\}$ is called the *local configuration* of $e \in E$. A *cut* is a maximal (w.r.t. set inclusion) 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 $\pi$. 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 $\Sigma$, denoted $Mark(\mathcal{C})$. A marking $M$ of $\Sigma$ is *represented* in $\pi$ if the latter contains a finite configuration $\mathcal{C}$ such that $M = Mark(\mathcal{C})$. Every marking represented in $\pi$ is reachable, and every reachable marking is represented in the unfolding of $\Sigma$.

A branching process $\pi = (B, E, G, h)$ of $\Sigma$ is *complete* if there is a set $E_{cut} \subseteq E$ of *cut-off* events such that for every reachable marking $M$ of $\Sigma$ there exist a finite configuration $\mathcal{C}$ of $\pi$ such that $\mathcal{C} \cap E_{cut} = \emptyset$ and $M = Mark(\mathcal{C})$, and for each such $\mathcal{C}$ and every transition $t$ enabled by $M$, there is an event $e \notin \mathcal{C}$ in $\pi$ such that $h(e) = t$ and $\mathcal{C} \cup \{e\}$ is a configuration ($e$ may be in $E_{cut}$).[2]

Although, in general, an unfolding is infinite, for every bounded net system $\Sigma$ one can construct a finite complete prefix $Pref_{\Sigma}$ of the unfolding of $\Sigma$, 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 $\Sigma$ augmented with an additional labelling of its events, $\lambda \circ h : E \to Z^{\pm} \cup \{\tau\}$. One can easily check the consistency of $\Gamma$, once its finite and complete prefix has been built ([15]).

## 3. State coding conflict detection using integer programming

Let $\Gamma = (\Sigma, Z, \lambda)$ be an STG, and $Unf_{\Gamma} \overset{\text{df}}{=} (B, E, G, \mathcal{M}_{in})$ be the safe net system built from a finite and complete prefix $Pref_{\Gamma} = (B, E, G, h)$ of the unfolding of $\Gamma$, fixed for the rest of this paper, where $\mathcal{M}_{in}$ is the canonical initial marking of $Unf_{\Gamma}$ which places a single token in each of the minimal conditions and no token elsewhere.[3] Furthermore, we will assume that $b_1, b_2, \ldots, b_p$ and $e_1, e_2, \ldots, e_q$ are respectively the conditions and events of $Pref_{\Gamma}$, and that $\mathcal{I}$ is the $p \times q$ incidence matrix of $Unf_{\Gamma}$. The set of cut-off events of $Pref_{\Gamma}$ will be denoted by $E_{cut}$.

Suppose that two distinct reachable markings $M'$ and $M''$ of $\Gamma$ are in CSC (or USC) conflict. Then these markings are represented in $Pref_{\Gamma}$ as some configurations $\mathcal{C}'$ and $\mathcal{C}''$ without cut-off events such that $M' = Mark(\mathcal{C}')$ and $M'' = Mark(\mathcal{C}'')$. Let $x'$ and $x''$ be respectively the Parikh vectors of $\mathcal{C}'$ and $\mathcal{C}''$. We will now transform the problem of checking for the presence of CSC (or USC) conflict into an integer programming problem expressed in terms of $x'$ and $x''$. Note that since these variables denote Parikh vectors of configurations, and no transition in $Unf_{\Gamma}$ can be fired more than once, they are from the domain $\{0,1\}^q$. The constraints constituting the system to be solved are described below.

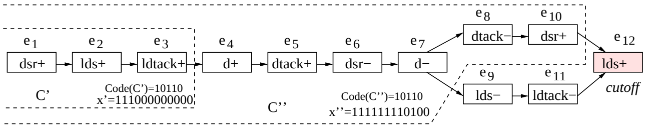

**Figure 2. Unfolding prefix for the VME bus example and CSC conflict between configurations $\mathcal{C}'$ and $\mathcal{C}''$ corresponding to markings $M'$ and $M''$ in Fig. 1(b). The corresponding Parikh vectors are $x'$ and $x''$.**

**Conflict constraints** For a configuration $\mathcal{C}$, its signal encoding vector $Code(\mathcal{C})$ can be expressed as

$$Code(\mathcal{C}) = v^0 + v^{\mathcal{C}} \ ,$$

where $v^0$ is the vector of the initial values of the signals and $v^{\mathcal{C}}$ is the signal change vector a configuration (defined similarly to $v^{\sigma}$, which was defined above for sequences). The above expression is a linear function of the Parikh vector $x^{\mathcal{C}}$ of $\mathcal{C}$; we will denote it by $Code(x^{\mathcal{C}})$. With this notation, the condition that $M'$ and $M''$ have the same signal encoding can be expressed as the linear constraint

$$Code(x') = Code(x'') \ . \qquad (2)$$

Note that the value of $v^0$ is not needed to build it.

Fig. 2 illustrates CSC conflict in the unfolding prefix of the STG shown in Fig. 1. Constraint (2) has the form:

$$\begin{cases} x'_1 - x'_6 + x'_{10} & = & x''_1 - x''_6 + x''_{10} & (dsr) \\ x'_5 - x'_8 & = & x''_5 - x''_8 & (dtack) \\ x'_2 - x'_9 + x'_{12} & = & x''_2 - x''_9 + x''_{12} & (lds) \\ x'_3 - x'_{11} & = & x''_3 - x''_{11} & (ldtack) \\ x'_4 - x'_7 & = & x''_4 - x''_7 & (d) \end{cases}$$

**Compatibility constraints** When solving the conflict constraint $Code(x') = Code(x'')$, we must ensure that the vectors $x'$ and $x''$ are indeed Parikh vectors of some configurations. This may be done in the following way. Since $Unf_{\Gamma}$ is an acyclic net, the feasibility of the marking equation $\mathcal{M} = \mathcal{M}_{in} + \mathcal{I} \cdot x$ is a necessary and sufficient condition for a marking $\mathcal{M}$ to be reachable. Therefore, if the system of inequalities $\mathcal{M}_{in} + \mathcal{I} \cdot x \geq \mathbf{0}$ has a non-negative integer solution then $x$ is a valid Parikh vector, and the Parikh vector of any configuration is a solution of this system (see [8, 14]). As we will see later in this paper, there is even better way to ensure that solutions are valid Parikh vectors, avoiding generation of these constraints.

**Cut-off constraints** In order to ensure that the configurations $\mathcal{C}'$ and $\mathcal{C}''$ contain no cut-off events, we add the following constraints:

$$x'(e) = 0 \text{ and } x''(e) = 0, \quad \text{for each } e \in E_{cut} \ . \qquad (3)$$

[2] This notion of completeness differs from the one given in [6], which does not mention cut-off events, and hence is not appropriate for algorithms making use of them. One can show that the unfolding algorithm proposed in [6] builds prefixes which are complete not only in the sense of the definition given [6], but also in the stronger sense assumed here.

[3] We will often identify $Unf_{\Gamma}$ and $Pref_{\Gamma}$, provided that this does not create an ambiguity.

The intuition is to prohibit the cut-off events from occurring by forcing the correspondent components of the Parikh vectors to be 0. This technique was used in [8, 14] in the context of checking reachability properties of Petri nets. Note that adding constraints of this type in fact reduces the system of constraints, effectively removing some of the variables. For the example of Fig. 2, such a constraint is $x'_{12} = x''_{12} = 0$, which simplifies the equation for *lds*.

**USC separating constraint** We are not interested in solutions with $M' = M''$, and so the constraint $M' \neq M''$ should be added. Together with (2) and (3), it provides a full formulation of the USC conflict detection problem using the integer programming framework (one should still ensure that compatibility constraints hold).

It is advantageous to make all the constraints in a system linear, since more good heuristics can be applied for solving it. The constraint $M' \neq M''$ does not fit this requirement, but we can replace it by $M' <_{lex} M''$, where $<_{lex}$ is the lexicographical order on markings of $\Gamma$. When $\Gamma$ is a bounded STG, this constraint is linear in the vectors $M' = (\mu'_1, \ldots, \mu'_n)$ and $M'' = (\mu''_1, \ldots, \mu''_n)$. Indeed, if every place of the STG can hold at most $k$ tokens then this constraint is equivalent to

$$\mu'_1 k^0 + \mu'_2 k^1 + \cdots + \mu'_n k^{n-1} < \mu''_1 k^0 + \mu''_2 k^1 + \cdots + \mu''_n k^{n-1},$$

and is very similar to the comparison of two $k$-ary numbers; in particular, when $\Gamma$ is safe, $M'$ and $M''$ can be seen as two binary numbers. Section 5 provides a way of rendering this constraint as a linear constraint specified for $x'$ and $x''$.

**CSC separating constraint** The separating constraint is more complex when we are interested in checking for the absence of CSC rather than USC conflicts. In general, it is hard to encode the constraint $Out(M') \neq Out(M'')$ as a linear constraint, therefore we would suggest to check for the presence of USC conflicts first. If there are none then CSC conflicts are also absent. If there is a USC conflict then in most cases it is also a CSC conflict. If there are USC conflicts which are not CSC conflicts, then a non-linear integer programming problem can be attempted. Though fewer heuristics work for non-linear systems, often a solution can be obtained in reasonable time.

To check if $Out(M') \neq Out(M'')$ holds for particular Parikh vectors $x'$ and $x''$ of configurations $\mathcal{C}'$ and $\mathcal{C}''$ one can compute $M' = Mark(\mathcal{C}')$ and $M'' = Mark(\mathcal{C}'')$ and find $Out(M')$ and $Out(M'')$ directly from the STG.

## 4. Integer programming verification algorithm

We have shown that CSC and USC conflict detection can be reduced to (possibly, non-linear) integer programming problems. In principle, at this point the standard solvers can be used to search for a solution. However, since they need too much time even for STGs of moderate size, a further refinement is needed.

In this section, we describe how solving the obtained system of constraints can be improved by taking into account partial-order dependencies between the variables, derived from the unfolding. For example, if we set $x(e) = 1$ then each $x(f)$ such that $f$ is a predecessor (in the causal order) of $e$ must be equal to 1, and each $x(g)$ such that $g$ is in conflict with $e$, must be equal to 0. Similarly, if we set $x(e) = 0$ then no event $f$ for which $e$ is a cause can be executed in the same run, and so $x(f)$ must be equal to 0. These observations can be formalised by considering $Unf_\Gamma$-compatible vectors (see section 2 for the definition), and the following result provides a basis for such an approach.

**Theorem 1** *A vector $x \in \{0,1\}^q$ is $Unf_\Gamma$-compatible iff for all distinct events $e, f \in E$ such that $x(e) = 1$, we have:*

$$f \prec e \Rightarrow x(f) = 1 \quad \text{and} \quad f \# e \Rightarrow x(f) = 0 \,. \tag{4}$$

**Corollary 1** *For each reachable marking $M$ of $\Sigma$, there exists an execution sequence of $Unf_\Gamma$ leading to a marking representing $M$, whose Parikh vector $x$ satisfies (4), and for every $e \in E_{cut}$, $x(e) = 0$.*

There exists a one-to-one correspondence between $Unf_\Gamma$-compatible vectors and configurations of the finite and complete prefix which was taken as the basis of $Unf_\Gamma$. In view of the last result, it is sufficient for our algorithm to check only $Unf_\Gamma$-compatible vectors whose components corresponding to cut-off events are equal to zero. This can be done by setting $x'(e) = x''(e) = 0$ for all $e \in E_{cut}$ at the beginning of the algorithm and constructing the *minimal $Unf_\Gamma$-compatible closure* (see below) of the current vector in each step of the algorithm.

**Definition 1** *A $Unf_\Gamma$-compatible vector $y \in \{0,1\}^q$ is a* $Unf_\Gamma$-compatible closure *of a vector $x \in \{0,1\}^q$ if $x \leq y$. Moreover, $y$ is the* minimal $Unf_\Gamma$*-compatible closure of $x$, denoted by $MCC(x)$, if it is minimal with respect to $\leq$ among all possible $Unf_\Gamma$-compatible closures of $x$.*

Note that $MCC(x)$ is undefined for some $x$'s, but whenever defined it is unambiguous due to Theorem 2 below.

**Theorem 2** *A vector $x \in \{0,1\}^q$ has a $Unf_\Gamma$-compatible closure iff for all $e, f \in E$, $x(e) = x(f) = 1$ implies $\neg(e \# f)$. If $x$ has a $Unf_\Gamma$-compatible closure then its minimal $Unf_\Gamma$-compatible closure exists and is unique. Moreover, in such a case if $x$ has zero components for all cut-off events, then the same is true for $MCC(x)$.*

From the implementation point of view, it may happen that a vector $x$ has an $\mathit{Unf}_\Gamma$-compatible closure according to Theorem 2, but it cannot be computed because some of the zero components of $x$ to be set to 1 have been fixed to 0 during the search process. In such a case, the algorithm should behave as if such a closure cannot be built.

One can see that the compatibility constraints are not essential for an algorithm checking only $\mathit{Unf}_\Gamma$-compatible vectors. Indeed, they are just the result of the substitution of $\mathcal{M} = \mathcal{M}_{in} + \mathcal{I} \cdot x$ into the constraints $\mathcal{M} \geq \mathbf{0}$ and hold for any $\mathit{Unf}_\Gamma$-compatible vector $x$ ([8]). Consequently, these inequalities may be left out without adding any $\mathit{Unf}_\Gamma$-compatible solution.

Various heuristics used by general purpose solvers can be implemented to reduce the search effort, especially when we terminate the search after finding one solution. The full version in [9] provides a background for this, as well as the implementation details of our algorithm.

## 5. Extended reachability analysis

In this section we show how an integer programming problem stated in terms of reachable markings of a Petri net can be expressed using $\mathit{Unf}_\Gamma$-compatible vectors.

Let us consider a property $P(M^{(1)}, \ldots, M^{(k)})$ specified for the markings of the original net system $\Sigma$. We can transform it into a corresponding property $\mathcal{P}(x^{(1)}, \ldots, x^{(k)})$ specified for $\mathit{Unf}_\Gamma$-compatible vectors $x^{(1)}, \ldots, x^{(k)}$ in such a way that if there exist reachable markings $\hat{M}^{(1)}, \ldots, \hat{M}^{(k)}$ of $\Sigma$ for which $P$ holds then $\mathcal{P}$ holds for some $\mathit{Unf}_\Gamma$-compatible vectors $\hat{x}^{(1)}, \ldots, \hat{x}^{(k)}$, and vice versa. Indeed, let $M^{(i)}$ be a reachable marking of $\Sigma$, and $\mathcal{M}^{(i)}$ be a corresponding marking in $\mathit{Unf}_\Gamma$. Then $M^{(i)}(s)$ can be expressed as

$$M^{(i)}(s) = \sum_{b \in h^{-1}(s)} \mathcal{M}^{(i)}(b) \ ,$$

where the marking $\mathcal{M}^{(i)}(b)$ of a place $b$ in $\mathit{Unf}_\Gamma$ can be found from the marking equation

$$\mathcal{M}^{(i)}(b) = \mathcal{M}_{in}(b) + \sum_{f \in {}^\bullet b} x^{(i)}(f) - \sum_{f \in b^\bullet} x^{(i)}(f) \ .$$

Therefore,

$$M^{(i)}(s) = \sum_{b \in h^{-1}(s)} \left( \mathcal{M}_{in}(b) + \sum_{f \in {}^\bullet b} x^{(i)}(f) - \sum_{f \in b^\bullet} x^{(i)}(f) \right) ,$$

and $P(M^{(1)}, \ldots, M^{(k)})$ can be rendered as a predicate $\mathcal{P}(x^{(1)}, \ldots, x^{(k)})$ specified for $\mathit{Unf}_\Gamma$-compatible vectors. And, moreover, if $P$ is initially expressed as a system of linear constraints then $\mathcal{P}$ will possess this property as well.

## 6. Verifying the normalcy property

The property of *normalcy* is a necessary condition for an STG to be implementable as a logic circuit built of gates whose characteristic functions are monotonic. The latter in turn guarantees that the circuit is speed-independent without the necessity to neglect (quite unrealistically) the delays in gates' input inverters (see [16]).

Let $\Gamma = (\Sigma, Z, \lambda)$ be an STG. Normalcy is specified with respect to an output signal $z \in Z_O$, and can be given in terms of the boolean *next-state* function $\mathit{Nxt}_z$ defined for the reachable markings. If $M$ is a reachable marking of $\Gamma$ and $u = \mathit{Code}(M)$, then: $\mathit{Nxt}_z(M) \overset{\text{df}}{=} 0$ if $u_z = 0$ and no $(z+)$–labelled transition is enabled in $M$, or $u_z = 1$ and a $(z-)$–labelled transition is enabled in $M$; and $\mathit{Nxt}_z(M) \overset{\text{df}}{=} 1$ if $u_z = 1$ and no $(z-)$–labelled transition is enabled in $M$, or $u_z = 0$ a $(z+)$–labelled transition is enabled in $M$.

$\Gamma$ satisfies the *positive normalcy* (or *p-normalcy*) condition w.r.t. an output signal $z$ if for every pair of reachable markings $M'$ and $M''$, $\mathit{Code}(M') \geq \mathit{Code}(M'')$ implies $\mathit{Nxt}_z(M') \geq \mathit{Nxt}_z(M'')$. Similarly, $\Gamma$ satisfies the *negative normalcy* (or *n-normalcy*) condition w.r.t. an output signal $z$ if for every pair of reachable markings $M'$ and $M''$, $\mathit{Code}(M') \geq \mathit{Code}(M'')$ implies $\mathit{Nxt}_z(M') \leq \mathit{Nxt}_z(M'')$. Finally, $\Gamma$ is *normal* if each output signal is either p-normal or n-normal. It turns out that normalcy implies CSC ([16]).

An example of normalcy violation is illustrated in Fig. 3. This is a state graph obtained after CSC conflict has been resolved in the STG of Fig. 1 by means of a new state signal, *csc*. The resulting model, free from CSC conflicts, is implementable — the next-state functions for all output signals are as follows: $lds = d + csc, dtack = d, d = ldtack \cdot csc$ and $csc = dsr \cdot (csc + \overline{ldtack})$. Nonetheless, normalcy is violated for signal *csc*. Indeed, $\mathit{Code}(M) > \mathit{Code}(M'), \mathit{Nxt}_{csc}(M) > \mathit{Nxt}_{csc}(M')$ but $\mathit{Code}(M) < \mathit{Code}(M''), \mathit{Nxt}_{csc}(M) > \mathit{Nxt}_{csc}(M'')$, so *csc* is neither n-normal nor p-normal. This is reflected in the implementation function for *csc*, which is non-monotonic: it is positive w.r.t. $dsr$ and negative w.r.t. $ldtack$ (note that the corresponding gate has an input inverter).

The verification method described earlier in this paper can be adopted to check the normalcy of STGs. Indeed, it is enough to solve in $\mathit{Unf}_\Gamma$-compatible vectors $x', x''$ the following (non-linear) system of constraints:

$$\begin{cases} \mathit{Code}(x') \geq \mathit{Code}(x'') \\ x'(e) = 0 \text{ and } x''(e) = 0 \quad \text{for each } e \in E_{cut} \\ \mathit{Nxt}_z(x') \; R_z \; \mathit{Nxt}_z(x'') \quad \text{for each } z \in Z_O \ , \end{cases} \tag{5}$$

where $R_z \in \{<, >\}$ depending on the type of normalcy of the signal $z$. If it is not known in advance, the algorithm can leave $R_z$ undefined until the moment when the first two

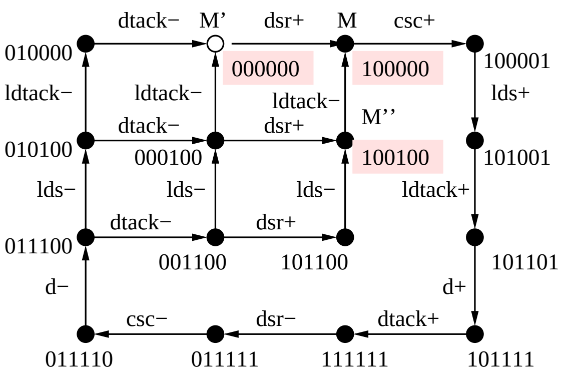

**Figure 3. State graph which is free from CSC conflicts but shows normalcy violations (for signal $csc$) between markings $M, M'$ and $M''$. The order of signals in the binary codes is: $dsr, dtack, lds, ldtack, d, csc$.**

constraints are satisfied and $Nxt_z(x') \neq Nxt_z(x'')$. Then it fixes $R_z$ so that the constraint does not hold and continues the search.

## 7. Dynamically conflict-free nets

In many cases the performance of our algorithm can be improved by exploiting specific properties of the Petri net underlying an STG $\Gamma$. For instance, if $\Gamma$ is free from dynamic conflicts (in particular, this is the case for marked graphs) then the union of any two configurations of $Unf_\Gamma$ is also a configuration. This observation can be used to reduce the search space. Indeed, according to proposition 1 below, it is enough to check only pairs of configurations $\mathcal{C}'$ and $\mathcal{C}''$ which are ordered in the set-theoretical sense.

**Proposition 1** *Let $\Gamma$ be free from dynamic conflicts, and let $\mathcal{C}'$ and $\mathcal{C}''$ be two finite configurations of $Unf_\Gamma$ such that $\mathcal{C}' \nsubseteq \mathcal{C}''$ and $\mathcal{C}'' \nsubseteq \mathcal{C}'$. If $Mark(\mathcal{C}')$ and $Mark(\mathcal{C}'')$ are in USC / CSC / p-normalcy / n-normalcy conflict, then the configuration $\mathcal{C} \overset{\text{df}}{=} \mathcal{C}' \cap \mathcal{C}''$ is such that $Mark(\mathcal{C})$ is in respectively USC / CSC / p-normalcy / n-normalcy conflict with either $Mark(\mathcal{C}')$ or $Mark(\mathcal{C}'')$.*

The conflict-freeness can be easily detected in $Unf_\Gamma$ and the above improvement can be incorporated into the algorithm ([9]).

## 8. Experimental results

The results of our experiments are summarised in Table 1. They were measured on a PC with $Pentium^{TM}$ III/500MHz processor and 128M RAM. 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; the number of conditions, events and cut-off events in the complete prefix; the time spent by a special version of the `Petrify` tool, which did not attempt to resolve the coding conflicts it had identified; and the time spent by our algorithm.

| Problem | Net | | | Unfolding | | | Time, [s] | |
|---|---|---|---|---|---|---|---|---|
| | $\|S\|$ | $\|T\|$ | $\|Z\|$ | $\|B\|$ | $\|E\|$ | $\|E_c\|$ | *Pfy* | *CLP* |
| LAZYRING | 35 | 32 | 11 | 87 | 66 | 5 | 2 | <0.01 |
| RING | 147 | 127 | 28 | 763 | 498 | 59 | 1498 | 0.01 |
| DUP-4PH-A | 133 | 123 | 26 | 144 | 123 | 11 | 36 | <0.01 |
| DUP-4PH-B | 135 | 123 | 26 | 146 | 123 | 11 | 35 | <0.01 |
| DUP-4PH-MTR-A | 109 | 96 | 22 | 117 | 96 | 8 | 25 | <0.01 |
| DUP-4PH-MTR-B | 114 | 105 | 26 | 122 | 105 | 8 | 25 | 0.02 |
| DUP-MOD-A | 129 | 100 | 21 | 199 | 132 | 10 | 222 | <0.01 |
| DUP-MOD-B | 116 | 165 | 21 | 344 | 218 | 65 | 623 | <0.01 |
| DUP-MOD-C | 152 | 115 | 27 | 228 | 149 | 13 | 286 | <0.01 |
| CF-SYM-A-CSC | 85 | 60 | 22 | 1341 | 720 | 56 | 357 | 107 |
| CF-SYM-B-CSC | 55 | 32 | 16 | 160 | 71 | 6 | 15 | 0.06 |
| CF-SYM-C-CSC | 59 | 36 | 18 | 286 | 137 | 10 | 33 | 2.30 |
| CF-SYM-D-CSC | 45 | 28 | 14 | 120 | 54 | 6 | 8.6 | 0.01 |
| CF-ASYM-A-CSC | 128 | 112 | 34 | 1808 | 1234 | 62 | 3988 | 2807 |
| CF-ASYM-B-CSC | 128 | 112 | 32 | 1816 | 1238 | 62 | 6144 | 3280 |

**Table 1. Experimental results: real-life STGs.**

The examples came from the real design practice and involve models of: ring protocol adapters ([1, 12]), duplex channel controller ([7]), and counterflow pipeline controller ([18]). Some STGs, although built by hand, were quite large in size. Other examples, including scalable ones, can be found in [9].

Note that in all cases the size of the complete prefix was relatively small. This may be explained by the fact that STGs usually contain a lot of concurrency but rather few conflicts, and thus the prefixes are not much bigger then the STGs themselves. As a result, the memory requirements of our algorithm are very moderate: it uses only $O(|E|)$ memory besides that needed to store the prefix, which for all the examples shown in the table means not more than just few kilobytes (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 our testing was limited in scope, we can draw some conclusions about the performance of the proposed algorithm. There is an important distinction between the ways how `Petrify` and `CLP` solve the problem of CSC checking: `Petrify` calculates the characteristic function of all CSC conflicts, whereas `CLP` can be stopped after the first CSC conflict has been detected. If a specification contains a coding conflict, our algorithm in most cases finds it very quickly. On the other hand, conflict-free specifications (bottom half of the table) are much harder to deal with. This may be explained by the fact that in such a case the algorithm has to explore the full search space. In the worst case, this may result in exploring all pairs of configurations. However, the heuristics we described allow the algorithm to considerably reduce the search space and produce the result in acceptable time.

## 9. Conclusions

Experimental results indicate that the algorithm proposed in this paper is not memory-demanding and in most cases time-efficient, though on some examples its performance was not entirely satisfactory. It is worth emphasising that the proposed approach overcomes the memory limitations of existing state-based methods, while still offering quite good performance.

### Acknowledgements

We would like to thank Jordi Cortadella for compiling a special version of the `Petrify` tool used in our experiments, and Alexander Letichevsky and Sergei Krivoi for drawing our attention to integer programming. Many thanks to the anonymous reviewers for their comments.

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.

## References

[1] C. Carrion and A. Yakovlev: Design and Evaluation of two Asynchronous Token Ring Adapters. Technical Report CS-TR-562, Dept. of Comp. Sci., Univ. of Newcastle (1996).

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

[3] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno and A. Yakovlev: `Petrify`: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Trans. on Inf. and Syst.* E80-D(3) (1997) 315–325.

[4] J. Cortadella, A. Kondratyev, M. Kishinevsky, L. Lavagno and A. Yakovlev: Complete state encoding based on theory of regions. Proc. of *ASYNC'96*, IEEE Comp. Soc. Press (1996) 36–47.

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

[6] J. Esparza, S. Römer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. Proc. of *TACAS'96*, Margaria T., Steffen B. (Eds.). Springer-Verlag, Lecture Notes in Computer Science 1055 (1996) 87–106.

[7] S. B. Furber, A. Efthymiou, and Montek Singh: A power-efficient duplex communication system. Proc. of *Int. Workshop on Asynchronous Interfaces: Tools, Techniques and Implementations*, Yakovlev A., Nouta R. (Eds.). TU Delft, The Netherlands (2000).

[8] V. Khomenko and M. Koutny: LP Deadlock Checking Using Partial Order Dependencies. Proc. of *CONCUR'2000*, Palamidessi C. (Ed.). Springer-Verlag, Lecture Notes in Computer Science 1877 (2000) 410–425.

[9] V. Khomenko, M. Koutny and A. Yakovlev: Detecting State Coding Conflicts in STGs Using Integer Programming. Technical Report. Technical Report CS-TR-736, Department of Computing Science, University of Newcastle (2000). see: www.cs.ncl.ac.uk/research/trs/lists/2001.html

[10] 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 *Int. Conf. Appl. of Conc. to Syst. Des. (CSD'98)*, IEEE Comp. Soc. Press (1998) 152–163.

[11] A. Kondratyev, J. Cortadella, M. Kishinevsky, E. Pastor, O. Roig and A. Yakovlev: Checking Signal Transition Graph Implementability by symbolic BDD traversal. Proc. of *EDTC'95*, IEEE Comp. Soc. Press (1995) 325–332.

[12] K. S. Low and A. Yakovlev: Token Ring Arbiters: an exercise in asynchronous logic design with Petri nets. Technical Report CS-TR-537, Dept. of Comp. Sci., Univ. of Newcastle (1995).

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

[14] S. Melzer and S. Römer: Deadlock Checking Using Net Unfoldings. Proc. of *Computer Aided Verification (CAV'97)*, O. Grumberg (Ed.). Springer-Verlag, Lecture Notes in Computer Science 1254 (1997) 352–363.

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

[16] N. Starodoubtsev, S. Bystrov, M. Goncharov, I. Klotchkov and A. Smirnov: Towards Synthesis of Monotonic Asynchronous Circuits from Signal Transition Graphs. Proc. of *ICACSD'01*, IEEE Comp. Soc. Press (2001) 179–188.

[17] P. Vanbekbergen, F. Catthoor, G. Goossens and H. De Man: Optimized Synthesis of Asynchronous Control Circuits form Graph-Theoretic Specifications. Proc. of *ICCAD'90*, IEEE CS Press (1990) 184–187.

[18] A. Yakovlev: Designing Control Logic for Counterflow Pipeline Processor Using Petri nets. *Formal Methods in Systems Design* 12(1) (1998) 39–71.