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 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) or Unique State Coding (USC) requirements, e.g., by using model-checking based on the reachability graph of an STG.

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 conflicts between the events involved in a finite and complete prefix of its unfolding. The model-checking algorithm is derived by adopting the integer programming approach. Following the basic formulation of the state coding conflict relationship, we present some problem-specific optimisation rules. This technique leads not only to huge memory savings when compared to the CSC (or USC) detection methods based on reachability graphs, but also to significant speedups in many cases. In addition, the method allows one to easily derive execution paths leading to an encoding conflict. Finally, the approach is also extended for checking the normalcy property of STGs, which is a necessary condition for their implementability using gates whose characteristic functions are monotonic.

Keywords: asynchronous circuits, automated synthesis, complete state coding, normalcy, Petri nets, signal transition graphs, integer programming, 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 ([4, 36]). STGs 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: (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 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` ([8]), performs all of these steps automatically, after first constructing the reachability 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.

To increase efficiency, `Petrify` uses 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 very convenient for finding good synthesis solutions, the issue of computational complexity for highly concurrent STGs is quite serious due to a combinatorial explosion of the state space. This puts practical bounds on the size of control circuits, which are sometimes 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 the state space explosion problem, Petri net analysis techniques based on causal partial order semantics, in the form of a Petri net unfoldings, had been applied for circuit synthesis ([31, 32]). 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 reachability 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 ([31]). The main shortcoming of the work described in [31, 32] 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 (see [21]). 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 this paper, we propose a solution for one of the sub-problems, central to the implementability analysis in step (a), viz. checking the Complete State Coding (CSC) and the Unique State Coding (USC) conditions ([4]). In essence, this problem consists in detecting the state coding conflicts, which occur when semantically different reachable states have the same binary encoding.

The CSC (and USC) problem is often seen as one which consists of two parts: the detection of coding conflicts, and the elimination of such conflicts. 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' to the system in the form of internal signals. The latter approach typically requires behaviour-preserving (with respect to the original set of events) transformations, which are more difficult than simply adding new ordering constraints. A number of methods for solving the CSC problem are available (see, e.g., [9], for a brief review). Most of them 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). Some, such as [35], operate directly on the STG level, but they restrict the class of the underlying Petri nets to, e.g., marked graphs.

The application of Petri Net unfoldings to the detection of state conflicts in an STG was first attempted in [21]. That work has advanced the ideas of slices and cover approximations of [31, 32], and presented 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 that evaluate a given boolean cover to true.

In this paper, we are investigating another kind of unfolding-based approach, involving integer programming. We show that the notion of state conflict can be characterised 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 [21], 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 ([19]), which has subsequently been extended to solving other model-checking problems ([18]). The results of initial experiments demonstrate that the proposed algorithm can in many cases achieve significant speedups. It is also worth pointing out that the method allows one not only to find conflicting reachable states, but also to derive execution paths leading to them without performing a reachability analysis.

The paper is organised as follows. In section 2 we provide the basic theoretical background concerning Petri nets, STGs, state conflicts, the CSC and USC problems and, in particular, net and STG unfoldings. Section 3 describes how the coding conflict detection problem can be

reduced to the feasibility test of a system of integer constraints. The algorithm we propose in this paper is developed specifically to exploit partial order dependencies between events in the unfolding of a Petri net, and is based on the Contejean and Devie's algorithm (*CDA*) for solving systems of linear constraints in non-negative integers ([1, 2, 5–7]). It is described in section 4, where we provide theoretical background and implementation details, as well as outline ways of reducing the number of variables and constraints in the original system presented in section 3. Section 5 describes an approach which allows one to render a problem specified in terms of a Petri net into a corresponding problem defined for its unfolding. Section 6 describes the implementation of our algorithm. Section 7 describes the normalcy checking problem, and shows how our algorithm can be adopted to solve it. Section 8 presents some additional heuristics, which can be used when the prefix of the net unfolding has no conflicts, as it is the case for, e.g., marked graphs. In section 9 the approach is generalised to deal with STGs containing silent transitions. Section 10 contains the results of experiments obtained for a number of benchmark examples.

2 Basic definitions

In this section, we first present basic definitions concerning Petri nets and STGs, and then recall (see also [11]) notions related to net unfoldings.

2.1 Petri nets

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

A *net system* is a pair $\Sigma \stackrel{\text{df}}{=} (N, M_0)$ 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 $M' \stackrel{\text{df}}{=} M - {}^\bullet t + t^\bullet$, 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. 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 Σ 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 $k = 1$.

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, which generate a finite alphabet $Z^\pm \stackrel{\text{df}}{=} Z \times \{+, -\}$ of *signal transition labels*, and $\lambda : T \to Z^\pm \cup \{\tau\}$ is a labelling function, where τ is a label indicating an internal (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. Signal transitions are associated with the actions which change the value of a particular signal. 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.

We associate with the initial marking of Γ a binary vector $v^0 \stackrel{\text{df}}{=} (v^0_1, \ldots, v^0_{|Z|}) \in \{0,1\}^{|Z|}$, where v^0_i corresponds to the signal $z_i \in Z$. Moreover, with a sequence of transitions σ we associate an integer *signal change vector* $v^\sigma \stackrel{\text{df}}{=} (v^\sigma_1, v^\sigma_2, \ldots, v^\sigma_{|Z|}) \in \mathbb{N}^{|Z|}$, 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*[1] if, for every reachable marking M, all firing sequences σ from M_0 to M have the same *encoding vector* $Code(M) \stackrel{\text{df}}{=} v^0 + v^\sigma$, 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 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 rising and falling labels z alternate in any firing sequence of Γ. In this paper it is assumed that all the STGs considered are consistent.[2]

The *state graph* of Γ is a tuple $SG_\Gamma \stackrel{\text{df}}{=} (S, A, s_0, Code)$ such that: $S \stackrel{\text{df}}{=} [M_0\rangle$ is the set of *states*; $A \stackrel{\text{df}}{=} \{M \stackrel{t}{\to} M' \mid M \in [M_0\rangle \wedge M[t\rangle M'\}$ is the set of *transitions*; $s_0 \stackrel{\text{df}}{=} M_0$ 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_Γ are in *USC conflict* if they are assigned the same code. Γ satisfies the *Unique State Coding (USC)* property if no two states of SG_Γ are in conflict.

It is often the case that the signals 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 logical gates in the circuit. The problem of logic synthesis consists in deriving 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 $Out(M) \stackrel{\text{df}}{=} \{z \in Z_O \mid \exists t \in T : M[t\rangle \wedge \lambda(t) = z\pm\}$ be the set of enabled output signals, for every reachable state M.

Γ satisfies the *Complete State Coding (CSC)* property if, for every pair of states in SG_Γ that have the same binary code, their sets of enabled output signals are the same; moreover, two distinct states are in *CSC conflict* if they have the same binary codes, but the sets of enabled output signals are different. Clearly, if Γ satisfies USC then it also satisfies CSC.

Until section 9, we will assume that the considered STGs contain no τ-labelled transitions.

2.3 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 Σ with a vector $(\mu_1, \ldots, \mu_m)$ such that $M(s_i) = \mu_i$, for all $i \leq m$. The *incidence matrix* of Σ is an $m \times n$ matrix $I = (I_{ij})$ such that, for all $i \leq m$ and $j \leq n$,

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

The *Parikh vector* of a finite sequence of transitions σ is a vector $x_\sigma = (x_1, \ldots, x_n)$ such that x_i is the number of the occurrences of t_i within σ, for every $i \leq n$. One can show that if σ 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:

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

[1] This is a somewhat simplified notion of consistency; see [31] 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 ([31]).

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 *Σ-compatible* if it is the Parikh vector of some execution sequence of Σ. 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 Σ. However, for the class of acyclic nets (in particular, net unfoldings), this equation provides an *exact* characterisation of the set of reachable markings ([28]) — the fact which is crucial for the approach proposed in this paper.

2.4 Branching processes

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 $ON \stackrel{\text{df}}{=} (B, E, G)$ 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 and 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 $h(E) \subseteq T$; 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 ([10]) 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 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.

2.5 Configurations and cuts

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 $[e] \stackrel{\text{df}}{=} \{f \mid f \preceq e\}$ 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 π. Then $Cut(\mathcal{C}) \stackrel{\text{df}}{=} (Min(ON) \cup \mathcal{C}^\bullet) \setminus {}^\bullet\mathcal{C}$ 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.

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 $\mathcal{C}$ 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).[3]

[3] This notion of completeness differs from the one given in [11], 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 [11] builds prefixes which are complete not only in the sense of the definition given [11], but also in the stronger sense assumed here.

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

The *branching process* of an STG $\Gamma = (\Sigma, Z, \lambda)$ is the a branching process of Σ augmented with an additional labelling of its events, $\lambda \circ h : E \to Z^{\pm} \cup \{\tau\}$. One can easily check the consistency of Γ, once its finite and complete prefix has been built (see [31]).

We extend the function *Code* to finite configurations of the branching process of Γ through $Code(\mathcal{C}) \stackrel{\text{df}}{=} Code(Mark(\mathcal{C}))$. Moreover, for any finite set of events $E' \subseteq E$, $v^{E'} \stackrel{\text{df}}{=} (v_1^{E'}, \ldots, v_{|Z|}^{E'}) \in \mathbb{N}^{|Z|}$ is the vector such that each $v_i^{E'}$ is the difference between the number of $(z+)$–labelled and $(z-)$–labelled events in E'. Note that if E' is a suffix of some configuration then $v^{E'} \in \{-1, 0, 1\}^{|Z|}$.

3 State coding conflict detection using integer programming

Let $\Gamma = (\Sigma, Z, \lambda)$ be an STG, and $Unf_\Gamma \stackrel{\text{df}}{=} (B, E, G, \mathcal{M}_{in})$ be the safe net system built from a finite and complete prefix $Pref_\Gamma = (B, E, G, h)$ of the unfolding of Γ fixed for the rest of this paper, where $\mathcal{M}_{in}$ is the canonical initial marking of Unf_Γ which places a single token in each of the minimal conditions and no token elsewhere.[4] 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_Γ. 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 Γ 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 $M' = Code(\mathcal{C}')$ and $M'' = Code(\mathcal{C}'')$. 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, they are from the domain $\{0, 1\}^q$. The constraints constituting the system to be solved are described below.

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. 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

$$Code(x') = Code(x'') \ . \tag{2}$$

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

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_Γ 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 [17–19, 28]). As we will see later in this paper, there is a better way to ensure that solutions are valid Parikh vectors, avoiding generation of these constraints.

[4] We will often identify Unf_Γ and $Pref_\Gamma$, provided that this does not create an ambiguity.

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:

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

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 [17–19, 27, 28] 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.

USC separating constraint We are not interested in solutions with $M' = M''$, and so the constraint $M' \neq M''$ need to be added. Such a constraint, together with (2) and (3), provides a full formulation of the USC conflict detection problem using the integer programming framework.

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 Γ. When Γ 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

$$\mu'_1 k^0 + \mu'_2 k^1 + \cdots + \mu'_n k^{n-1} < \mu''_1 k^0 + \mu''_2 k^1 + \cdots + \mu''_n k^{n-1}, \tag{4}$$

and is very similar to the comparison of two k-ary numbers; in particular, when Γ 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. Not all the described in this paper heuristics work for non-linear systems, but 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 the systems of constraints to be solved usually large even for STGs of moderate size, a further refinement is needed.

In this section, we describe how solving such a system may be improved by taking into account partial-order dependencies between the variables, derived from the unfolding. After that, we outline an extension of the Contejean and Devie's algorithm (*CDA*), described in [5–7], aiming at combining these dependencies with the original algorithm.

The proofs of all the results can be found in [18, 20].

4.1 Partial-order dependencies between variables

During the solving of the constraint system, one may use dependencies between variables implied by the causal order on events, which can easily be derived from Unf_Γ. 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_Γ-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_Γ-compatible iff for all distinct events $e, f \in E$ such that $x(e) = 1$, we have:*

$$f \prec e \Rightarrow x(f) = 1 \quad \text{and} \quad f \# e \Rightarrow x(f) = 0\,. \tag{5}$$

Note: Unf_Γ-compatible vectors are binary, since each event in the unfolding of Σ can occur at most once in an execution sequence.

Corollary 1. *For each reachable marking M of Σ, there exists an execution sequence of Unf_Γ leading to a marking representing M, whose Parikh vector x satisfies (5), and for every $e \in E_{cut}$, $x(e) = 0$.*

There exists a one-to-one correspondence between Unf_Γ-compatible vectors and configurations of the finite and complete prefix which was taken as the basis of Unf_Γ. In view of the last result, it is sufficient for our algorithm to check only Unf_Γ-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_Γ-compatible closure* (see below) of the current vector in each step of the algorithm.

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

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

Theorem 2. *A vector $x \in \{0,1\}^q$ has a Unf_Γ-compatible closure iff for all $e, f \in E$, $x(e) = x(f) = 1$ implies $\neg(e \# f)$. If x has a Unf_Γ-compatible closure then its minimal Unf_Γ-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 Unf_Γ-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 Unf_Γ-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 Unf_Γ-compatible vector x ([17–19]). Consequently, these inequalities may be left out without adding any Unf_Γ-compatible solution.

4.2 An extension of CDA

In this section, we will present a general result extending that in [7]. Consider the following homogeneous system of linear constraints:

$$\begin{cases} \mathcal{A} \cdot x = \mathbf{0} \\ \mathcal{B} \cdot x \leq \mathbf{0} \\ x \in D \stackrel{\text{df}}{=} D_1 \times \cdots \times D_q\,, \end{cases} \tag{6}$$

where $D_i \stackrel{\text{df}}{=} \{k_i, k_i+1, \ldots, k_i+l_i\}$ and $k_i, l_i \geq 0$, for every $i \in \{1 \ldots q\}$. Below we assume that $\mathbf{0} \notin D$.[5]

Let $\xi : D \to D$ be a partial function[6] with the domain *dom* such that $x_{min} \stackrel{\text{df}}{=} (k_1, \ldots, k_q) \in dom$, and $codom \stackrel{\text{df}}{=} \xi(dom)$. A *$\xi$-minimal solution* of (6) is any solution $x \in codom$ for which there is no solution $y \in codom$ satisfying $y < x$. We will denote this by $x \in min_\xi$, and assume that:

$$\begin{array}{c} y \in dom \Longrightarrow y \leq \xi(y) \\ y \leq x \in min_\xi \Longrightarrow y \in dom \ \wedge \ \xi(y) \leq x \ . \end{array} \tag{7}$$

The aim is to develop an algorithm searching for all ξ-minimal solutions. First, we introduce a new modification of CDA's branching condition (see also [1, 2, 18, 20]).

Branching Condition 1 *A vector $x \in codom$ which is not a solution of (6) can be extended to $\xi(x+\varepsilon_j)$ if $x+\varepsilon_j \in dom$ and*

$$(\mathcal{A} \cdot x) \odot (\mathcal{A} \cdot \varepsilon_j) + \sum_{i=1}^{m} \min \left\{ \begin{array}{c} \max\{0, \mathcal{B}_i \odot x\}(\mathcal{B}_i \odot \varepsilon_j), \\ (\mathcal{B}_i \odot x)(\mathcal{B}_i \odot \varepsilon_j) \end{array} \right\} < 0 \ , \tag{8}$$

where m is the number of rows in $\mathcal{B}$, $\mathcal{B}_i$ is the i-th row of $\mathcal{B}$, ε_j is the j-th vector in the canonical basis CanBase of $\mathbb{N}^q$, and $\odot$ denotes the scalar product of two vectors. $\diamond$

The above rule determines a search space which can be represented by a labelled directed graph $G_\xi \stackrel{\text{df}}{=} (X, A)$, where $X \subseteq codom$ is a set of vertices and $A \subseteq X \times CanBase \times X$ is a set of arcs. It is defined as the smallest graph such that X contains a distinguished vertex $x_{root} \stackrel{\text{df}}{=} \xi(x_{min})$ and, for every $x \in X$ which is not a solution of (6), if $\varepsilon_j \in CanBase$ satisfies $x+\varepsilon_j \in dom$ and (8), then $(x, \varepsilon_j, \xi(x+\varepsilon_j)) \in A$. Directly from the definitions we obtain

Proposition 1. *G_ξ is finite and acyclic.*

The next proposition states a crucial property of the new branching condition.

Proposition 2. *If a vertex x of G_ξ and $y \in min_\xi$ satisfy $x < y$, then there is an arc (x, ε_j, z) in G_ξ such that $z \leq y$.*

Corollary 2. *All ξ-minimal solutions are vertices of G_ξ.*

Although the above corollary and Proposition 1 imply that G_ξ could be used to solve the problem at hand,[7] it may contain a large number of redundant paths. We will now adapt the frozen components method of [2, 7] to cope with this problem. Below, for any node x of G_ξ we denote by $out(x)$ the set of all the ε_j's which label the arcs outgoing from x.

Frozen Components 1 *We assume that, for each node x of G_ξ, there is a total ordering $\prec_x$ on the set $out(x)$. And, if $\varepsilon_i \prec_x \varepsilon_j$, then ε_j is* frozen *along all the directed paths in G_ξ beginning with the arc $(x, \varepsilon_i, \xi(x+\varepsilon_i))$.* $\diamond$

To capture the above rule through a suitable modification of G_ξ, we associate sets of frozen components with the arcs of directed paths originating at x_{root}. Let $\sigma = \alpha_1\alpha_2 \ldots \alpha_k$ be a sequence of arcs in G_ξ forming a directed path starting at x_{root}. For every arc $\alpha_i = (x, \varepsilon_j, y)$ in σ, we denote by $Froz_\sigma(\alpha_i)$ a subset of *CanBase* such that

$$Froz_\sigma(\alpha_i) \stackrel{\text{df}}{=} \{\varepsilon_m \in out(x) \mid \varepsilon_j \prec_x \varepsilon_m\} \cup \left\{ \begin{array}{ll} \emptyset & \text{if } i = 1 \\ Froz_\sigma(\alpha_{i-1}) & \text{if } i > 1 \ . \end{array} \right.$$

[5] From the point of view of this paper, such an assumption is unproblematic. The case $\mathbf{0} \in D$ is discussed in Remark 1, at the end of this section.

[6] Later in this section we will take ξ to be the *MCC* function.

[7] E.g., G_ξ could be searched in the breadth-first or depth-first manner.

We then say that σ is *non-frozen* if, for every arc $\alpha_i = (x, \varepsilon_j, y)$ in σ, $Supp(y-x) \cap Froz_\sigma(\alpha_i) = \emptyset$, where $Supp(x) \stackrel{\text{df}}{=} \{\varepsilon_j \mid \varepsilon_j \leq x\}$.

With the above notation, Frozen Components 1 determines a search space which can be represented by the smallest subgraph T_ξ of G_ξ containing x_{root} and all the non-frozen directed paths of G_ξ.

Theorem 3. *T_ξ is a tree rooted at x_{root} whose set of vertices contains all ξ-minimal solutions.*

We observe that since T_ξ is a tree, in the notation $Froz_\sigma(\alpha)$ we can drop the index σ (see the definition of $Froz_\sigma$).

The above frozen components rule allows for further improvement, which is given in the form of an additional function *froz*, which will be used to 'freeze' the variables corresponding to the events which are in conflict whit the current configuration being considered by the algorithm, so that they cannot be set to 1.

Frozen Components 2 *We assume that, for every arc α of T_ξ, $froz(\alpha)$ is a subset of CanBase such that if α and α' form two consecutive arcs then $froz(\alpha) \subseteq froz(\alpha')$. Moreover, if $\alpha_1 \ldots \alpha_k$ is a directed path in T_ξ leading from x_{root} to $y \in min_\xi$, then for every $i \leq k$, $froz(\alpha_i) \cap Supp(y - x_i) = \emptyset$, where x_i is the origin of α_i.* $\diamond$

Theorem 4. *Let S_ξ be the minimal subtree of T_ξ which contains x_{root} and all the directed paths non-frozen w.r.t. froz. Then the set of vertices of S_ξ comprises all ξ-minimal solutions.*

To summarise, Branching Condition 1 and Frozen Components 1 and 2 define a search tree which can be traversed[8] to find all ξ-minimal solution of (6) in a finite number of steps (as G_ξ is finite, see Proposition 1). The resulting approach can then be applied to deal also with a non-homogeneous system of linear constraints

$$\begin{cases} \mathcal{A} \cdot x = a \\ \mathcal{B} \cdot x \leq b \\ x \in D \end{cases} \tag{9}$$

where we *do not* assume that $\mathbf{0} \notin D$, and all the other notions and assumptions relating to ξ are as those for (6).

The problem of finding all ξ-minimal solutions of (9) can be reduced to an instance of the problem considered earlier in this section. To this end, we introduce an auxiliary variable z and two matrices, $\mathcal{A}' \stackrel{\text{df}}{=} (\mathcal{A}, -a)$ and $\mathcal{B}' \stackrel{\text{df}}{=} (\mathcal{B}, -b)$. Then (9) can be rewritten as

$$\begin{cases} \mathcal{A}' \cdot (x, z) = \mathbf{0} \\ \mathcal{B}' \cdot (x, z) \leq \mathbf{0} \\ (x, z) \in D' \stackrel{\text{df}}{=} D \times \{1\} \ . \end{cases} \tag{10}$$

Moreover, after setting $dom' \stackrel{\text{df}}{=} dom \times \{1\}$ and $\xi'(x, z) \stackrel{\text{df}}{=} (\xi(x), 1)$, we obtain an instance of (6) (note that $\mathbf{0} \notin D'$). We now observe that x is a ξ-minimal solution of (9) iff $(x, 1)$ is a ξ'-minimal solution of (10). As a result, we can render the branching condition derived for (10), directly in terms of (9).

Branching Condition 2 *A vector $x \in codom$ which is not a solution of (9) can be extended to $\xi(x + \varepsilon_j)$ if $x + \varepsilon_j \in dom$ and*

$$(\mathcal{A} \cdot x - a) \odot (\mathcal{A} \cdot \varepsilon_j) + \sum_{i=1}^{m} r_i < 0 \ , \tag{11}$$

where, for every $1 \leq i \leq m$, $r_i \stackrel{\text{df}}{=} 0$ if $\mathcal{B}_i \odot x \leq b_i$ and $\mathcal{B}_i \odot \varepsilon_j \leq 0$; otherwise $r_i \stackrel{\text{df}}{=} (\mathcal{B}_i \odot x - b_i)(\mathcal{B}_i \odot \varepsilon_j)$. $\diamond$

[8] Using the depth-first search as the breadth-first search would be inefficient due to the need of recording frozen components.

Remark 1. We assumed that $x_{min} \in dom$ since otherwise there are no ξ-minimal solutions at all.

To obtain a full extension of *CDA*, we still need to consider (6) when $\mathbf{0} \in D$ (note that $\mathbf{0}$ is a trivial solution and has to be excluded from the search). Our discussion can easily be adapted, as follows:

- We assume that $\mathbf{0} \notin codom$.
- $x_{root} \stackrel{\text{df}}{=} \mathbf{0}$, and if $\varepsilon_j \in dom$ then $(\mathbf{0}, \varepsilon_j, \xi(\varepsilon_j)) \in A$.

Then all the results presented earlier in this section still hold, in particular, Theorems 3 and 4.

Allowing infinite ranges $D_i \stackrel{\text{df}}{=} \{k_i, k_i+1, \ldots\}$ leads to termination problems; in other words, the search graph G_ξ may be infinite. In such a case, one needs to develop conditions for bounding ξ-minimal solutions. Such a problem depends on the actual definition of the function ξ, and so we expect that it will be addressed on the individual basis. In our case all variables are binary, so the termination is guaranteed. ◇

4.3 Applying the method for Unf_Γ-compatible vectors

We will now apply the theory developed in the previous section to check only Unf_Γ-compatible vectors. Referring to the notation introduced above, we shall assume that the system of constraints to be solved is a non-homogeneous one, and:

- $x'_i, x''_i \in D_i$, where $D_i \stackrel{\text{df}}{=} \{0\}$ if $e_i \in E_{cut}$, and $D_i \stackrel{\text{df}}{=} \{0,1\}$ otherwise.
- *dom* is the set of all vectors (x', x'') such that $x'_i, x''_i \in D_i$, both x' and x'' have Unf_Γ-compatible closures, and $\xi((x', x'')) \stackrel{\text{df}}{=} (MCC(x'), MCC(x''))$.
- For an arc $\alpha = ((x', x''), \varepsilon_j, (y', y''))$,

$$froz(\alpha) \stackrel{\text{df}}{=} \{\varepsilon_i \mid \exists \varepsilon_k \in Supp(y') : e_k \# e_i)\} \cup \{\varepsilon_{i+|E|} \mid \exists \varepsilon_k \in Supp(y'') : e_k \# e_i)\}$$

It is straightforward to show that all the properties required for *dom*, ξ and *froz* are then satisfied, and so after ignoring the auxiliary variable z, the search tree S_ξ contains all minimal Unf_Γ-compatible solutions.

Optimisations Various heuristics used by general purpose *MIP* solvers can be implemented to reduce the search effort, especially when we terminate the search after finding one solution.

For example, one can look one step ahead and choose a branch that in some sense is the 'most promising' one. This can be done by choosing an ordering on the sons of each node of the search tree, depending on the current value of x (e.g., the $\prec_{\|\cdot\|}$ ordering in [7]).

Moreover, if the algorithm, having fixed the values of some variables, finds out that some of the inequalities have become infeasible, then it may prune the current branch of the search tree. In addition, it is sometimes possible to determine the values of some variables which have not yet been fixed, or to find out that some of the constraints have become redundant (in [17], some simple heuristics of this sort were described).

After fixing the value of a variable, it is necessary to build the minimal Unf_Γ-compatible closure of the current vector. As new variables can become fixed during this process, the above tests can be applied iteratively. If the minimal Unf_Γ-compatible closure cannot be built due to frozen components, then the current subtree of the search tree contains no Unf_Γ-compatible solution and may be pruned. Such optimisation rules can formally be justified in the following way.

Let $opt : D \to D$ be a partial function[9] with the domain dom_{opt}, corresponding to applying the heuristics described above, satisfying:

$$\begin{array}{c} x \in dom_{opt} \Longrightarrow x \leq opt(x) \\ x \leq y \in min_\xi \Longrightarrow x \in dom_{opt} \ \wedge \ opt(x) \leq y \ . \end{array} \tag{12}$$

[9] Intuitively, $opt(x)$ is undefined if, during the application of the optimisation rules, the algorithm finds out that the system has no ξ-minimal solution $y \geq x$.

We then define a partial function $\xi_o : D \to D$ such that $\xi_o(x) \stackrel{\text{df}}{=} \xi(opt(\xi(x)))$, for every x in dom_o which is the largest subset of dom for which this expression is well-defined. We denote $codom_o \stackrel{\text{df}}{=} \xi_o(dom_o)$, and then observe that, by (7) and (12):

$$\begin{array}{c} x \in dom_o \Longrightarrow x \leq \xi_o(x) \\ x \leq y \in min_\xi \Longrightarrow x \in dom_o \ \wedge \ \xi_o(x) \leq y \ . \end{array} \tag{13}$$

Proposition 3. $min_\xi = min_{\xi_o}$.

From Proposition 3 and (13) it follows that the counterpart of (7) holds for ξ_o as well. Thus, in view of $min_\xi = min_{\xi_o}$, the search for ξ-minimal solutions can be based on the tree S_{ξ_o}, which can often be much smaller than S_ξ. As to the frozen components given by the function $froz_o$, it must satisfy Frozen Components 2 condition.

In order to avoid calculations related to redundant constraints, we can remember for each of them the depth in the search tree at which it was marked as redundant, and unmark it during the backtracking. Clearly, they do not need to be considered when checking whether the system is satisfied. What is more, the algorithm may skip them when computing the branching condition (see [18]).

5 Extended reachability analysis

The algorithm described in section 4.3 is applicable to any system of linear constraints which are supposed to be solved in Unf_Γ-compatible vectors. The theory of verifying *co-linear* properties using unfoldings was developed in [27]. It can easily be generalised to arbitrary reachability properties, although solving (sometimes very large) non-linear systems obtained in this case is usually a hard task for general-purpose solvers. An algorithm checking only Unf_Γ-compatible vectors can do this more efficiently. Indeed, the only reason why the algorithm in section 4.3 accepts only systems of linear constraints is that in order to reduce the search space it employs the branching condition (11). In principle, it can deal with arbitrary constraints, if one switches off this heuristic.

The approach we will now describe is similar to the one described in [27], generalised to deal with non-linear constraints. In addition, we use the ideas from section 4 to re-formulate the resulting integer programming problem in terms of Unf_Γ-compatible vectors.

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

Input
$Cons$ — a system of constraints
Unf_Γ — a net system built from $Pref_\Gamma$

Output
A Unf_Γ-compatible solution of $Cons$, if there is one

Initialisation
$depth \leftarrow 1$
$X', X'' \leftarrow (0, \ldots, 0)$
for $i \in \{1, \ldots, q\}$: $X'_{fix}[i], X''_{fix}[i] \leftarrow \begin{cases} 1 \text{ if } e_i \in E_{cut} \\ 0 \text{ otherwise} \end{cases}$

Main procedure

if (X', X'') is a solution of $Cons$
then return (X', X'')

for all $(i, X) \in \{1, \ldots, q\} \times \{X', X''\}$ **such that** $X_{fix}[i] = 0 \wedge (11)$ holds
$depth \leftarrow depth + 1$
if $set(i, X)$ has succeeded **then**
apply the procedure to the new values X', X'' recursively
if solution is found
then return (X', X'')

undo changes in the elements $X[j]$ such that $X_{fix}[j] = depth$
$depth \leftarrow depth - 1$
$freeze(i, X)$ /* never fails here as X is Unf_Γ-compatible */

return *'no solutions'*

Fig. 1. Integer programming verification algorithm.

The algorithm in figure 1 is very similar to that of CDA modified to solve non-homogeneous systems of linear constraints,[10] and checking only Unf_Γ-compatible vectors x' and x'', such that $x'(e) = x''(e) = 0$ for all $e \in E_{cut}$.

In the general case, the maximal depth of the search tree is $O(|E|)$, so the original CDA would need to store $O(|E|)$ vectors of length $|E|$. In our algorithm, we use just four arrays of length $|E|$:

- $X', X'' : array[1..q]\ of\ \{0, 1\}$
  To construct a solution.
- $X'_{fix}, X''_{fix} : array[1..q]\ of\ integers$
  To keep the information about the levels of fixing of the components of X' and X''.

The interpretation of these arrays is as follows (we denote by X either X' or X'', and by X_{fix} the corresponding array X'_{fix} or X''_{fix}):

- $X_{fix}[i] = 0$. Then $X[i]$ must be 0 and this means that $X[i]$ has not yet been considered, and may later be set to 1 or frozen.

[10] In order to solve non-linear systems it is enough to remove the restriction '$\wedge(11)$ holds' from the header of the **for all** loop of the algorithm.

- $X_{fix}[i] = k > 0$ and $X[i] = 0$. Then $X[i]$ has been frozen at some node on level k whose subtree the algorithm is developing. It cannot be unfrozen until the algorithm backtracks to level k.
- $X_{fix}[i] = k > 0$ and $X[i] = 1$. Then $X[i]$ has been set to 1 at some node on level k whose subtree the algorithm is developing. This value is fixed for the entire subtree.

Notice that storing the levels of fixing of the elements of X allows one to undo changes during backtracking, without keeping all the intermediate values of X.
We also use the following auxiliary variables and functions:

- $depth : integer$
  The current depth in the search tree.
- $freeze(i, X)$
  Freezes all $X[k]$'s such that $e_i \preceq e_k$. If there is $X[k] = 1$ to be frozen then *freeze* fails. The corresponding elements of X_{fix} are set to the current value of *depth*.
- $set(i, X)$
  Sets all $X[j]$'s such that $e_j \preceq e_i$ to 1 and uses *freeze* to freeze all $X[k]$'s such that $e_i \# e_k$. If there is a frozen $X[j]$ to be set to 1, or $X[k] = 1$ to be frozen then *set* fails. The current value of *depth* is written in the elements of X_{fix}, corresponding to the components being fixed.

Retrieving a solution What we often want to see as a solution is two execution sequences of the original STG, leading to states which are in CSC (or USC) conflict. Unlike BDD-based methods, in our algorithm we can obtain this information for free. Indeed, to derive such sequences, it is enough to topologically sort the constructed configurations according to the causal order on the set of the events, and replace the events by their labels in the constructed sequence. A relevant observation one can make is that the unfolders add events one-by-one to the unfolding being constructed, in such a way that for all non-cut-off events e_i and e_j, $e_i \prec e_j$ implies $i < j$. Therefore, if we use the natural numbering of the components of the vector x, $x_i = x(e_i)$, then we can avoid sorting the events and find the sequences of transitions in a straightforward way.

Short trails Finding short paths leading to states which are in CSC (or USC) conflict can facilitate the debugging of a circuit modelled by an STG. In such a case, we need to solve an optimisation problem with the same system of constraints, and $\mathcal{L}(x', x'') \stackrel{\text{df}}{=} \sum_{i=1}^{|E|}(x'_i + x''_i)$ as the cost function to be minimised.

The algorithm can easily be adopted for this task. We do not stop after the first solution has been found, but keep the current optimal solution together with the corresponding value of the function $\mathcal{L}$. As this function is non-decreasing, we may prune a branch of the search tree as soon as value of $\mathcal{L}$ becomes greater than, or equal to, the current optimal value. This strategy speeds up the search and saves us from keeping all ξ-minimal solutions found so far. It is easy to see that this strategy does not affect the completeness in the sense that a ξ-minimal solution minimising $\mathcal{L}$ can be computed if it exists. Indeed, the strategy builds the same search tree up to the cutting of some of the subtrees rooted in nodes with the sum of the components not less than the optimal value of $\mathcal{L}$. But all the descendants of such nodes have even greater sum of the components, and so these subtrees cannot contain an optimal solution. To allow more pruning and, therefore, to reduce the search space, it makes sense to organise the search process in such a way that the first solutions found give the value of $\mathcal{L}$ 'close' to the optimal one. This can be done by choosing in each step of the algorithm the 'most promising' branches. Since the orderings on the sons of each node of the search graph, used by the algorithm, are arbitrary, we may exploit the information about the value of $\mathcal{L}$ on them, and check successors with smaller values first (see, e.g., the $\prec_{\|\cdot\|}$ ordering in [7]). Such an algorithm can be seen as a version of the 'branch and bound' method which considers only Unf_Γ-compatible vectors and uses frozen components and branching condition to reduce the search space.

7 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 input inverters (see [34]).

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 Nxt_z defined for the reachable markings. If M is a reachable marking of Γ and $u = Code(M)$, then: $Nxt_z(M) \stackrel{\text{df}}{=} 0$ 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 $Nxt_z(M) \stackrel{\text{df}}{=} 1$ 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.

Γ 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'', $Code(M') \geq Code(M'')$ implies $Nxt_z(M') \geq Nxt_z(M'')$. Similarly, Γ 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'', $Code(M') \geq Code(M'')$ implies $Nxt_z(M') \leq Nxt_z(M'')$. Finally, Γ is *normal* if each output signal is either p-normal or n-normal. It turns out that normalcy implies CSC ([34]).

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

$$\begin{cases} Code(x') \geq Code(x'') \\ x'(e) = 0 \text{ and } x''(e) = 0 \quad \text{for each } e \in E_{cut} \\ Nxt_z(x') \; R_z \; Nxt_z(x'') \quad \text{for each } z \in Z_O \; , \end{cases} \tag{14}$$

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 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.

8 The case of 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 Γ. For instance, if Γ is free from dynamic conflicts (in particular, this is the case for marked graphs) then the union of any two configurations of Unf_Γ is also a configuration. This observation can be used to reduce the search space. Indeed, according to proposition 4 below, it is then enough to look only for those cases when the configurations $\mathcal{C}'$ and $\mathcal{C}''$ being tested are ordered in the set-theoretical sense.

Proposition 4. *Let Γ be free from dynamic conflicts, and let $\mathcal{C}'$ and $\mathcal{C}''$ be two finite configurations of Unf_Γ such that $\mathcal{C}' \not\subseteq \mathcal{C}''$ and $\mathcal{C}'' \not\subseteq \mathcal{C}'$. If $Mark(\mathcal{C}')$ and $Mark(\mathcal{C}'')$ are in USC / CSC / p-normalcy / n-normalcy conflict, then the configuration $\mathcal{C} \stackrel{\text{df}}{=} \mathcal{C}' \cap \mathcal{C}''$ is such that $Mark(\mathcal{C})$ is in respectively USC / CSC / p-normalcy / n-normalcy conflict with $Mark(\mathcal{C}')$ or $Mark(\mathcal{C}'')$.*

Proof. We first show that

$$Code(\mathcal{C}') \geq Code(\mathcal{C}'') \implies Code(\mathcal{C}') \geq Code(\mathcal{C}) \geq Code(\mathcal{C}'') \; . \qquad (*)$$

Since Γ is conflict-free, $\mathcal{C}'$, $\mathcal{C}''$ and $\mathcal{C}$ are all included in the configuration $\mathcal{C}' \cup \mathcal{C}''$, and so each event in $E' \stackrel{\text{df}}{=} \mathcal{C}' \setminus \mathcal{C}$ is concurrent to each event in $E'' \stackrel{\text{df}}{=} \mathcal{C}'' \setminus \mathcal{C}$. Now, due to the consistency of Γ, no two distinct concurrent events in Unf_Γ can have the same signal label. Hence none of the events in E' can have the same signal label (even after ignoring the sign) as an event in E''.

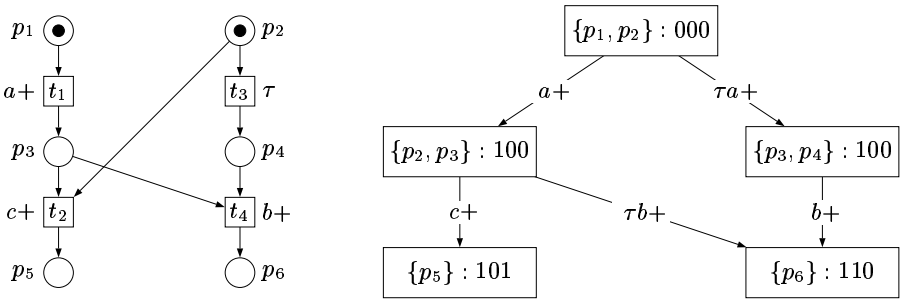

Fig. 2. An STG and its state graph with dummies playing a confusing role, where $z \in Z_O$.

Consequently, $v^{E'} \odot v^{E''} = \mathbf{0}$. We next observe that $Code(\mathcal{C}) + v^{E'} = Code(\mathcal{C}') \geq Code(\mathcal{C}'') = Code(\mathcal{C}) + v^{E''}$ implies that $v^{E'} \geq v^{E''}$. Together with $v^{E'} \odot v^{E''} = \mathbf{0}$ and the fact that both $v^{E'}$ and $v^{E''}$ belong to $\{-1, 0, +1\}^{|Z|}$, this leads to $v^{E'} \geq \mathbf{0} \geq v^{E''}$, and so $(*)$ holds.

By $(*)$, we immediately obtain that $Code(\mathcal{C}') = Code(\mathcal{C}'')$ implies $Code(\mathcal{C}') = Code(\mathcal{C}) = Code(\mathcal{C}'')$, and hence the proposition holds for the USC and CSC conflicts.

Suppose now that $Mark(\mathcal{C}')$ and $Mark(\mathcal{C}'')$ are, without loss of generality, in p-normalcy conflict due to $Code(\mathcal{C}') \geq Code(\mathcal{C}'')$ and $Nxt_z(\mathcal{C}') < Nxt_z(\mathcal{C}'')$ which implies that $Nxt_z(\mathcal{C}') = 0$ and $Nxt_z(\mathcal{C}'') = 1$. Then, by $(*)$, if $Nxt_z(\mathcal{C}) = 0$ then $Mark(\mathcal{C})$ and $Mark(\mathcal{C}'')$ are in p-normalcy conflict; otherwise, $Nxt_z(\mathcal{C}) = 1$ and $Mark(\mathcal{C})$ and $Mark(\mathcal{C}')$ are in p-normalcy conflict. □

In order to make the algorithm consider only ordered configurations, it is enough to choose an appropriate function $\hat{\xi}(x', x'') \stackrel{\text{df}}{=} (MCC(x'), MCC(x'|x''))$, where '|' is the bit-wise 'or' operation on binary vectors (see section 4.2). Note that if we are verifying the normalcy property using this improvement, one cannot say in advance whether $\mathcal{C}' \subset \mathcal{C}''$ or $\mathcal{C}'' \subset \mathcal{C}'$ in the solution, and so the algorithm has to check both cases.

9 Dealing with STG containing dummy events

So far our approach was applied to STGs without 'dummies', i.e., τ–labelled transitions, which are unrelated to any signal value changes.

The presence of dummy events may sometimes create problems in analysing state coding conditions due to the inherent ambiguity involved in their interpretation. Indeed, the binary encodings of a pair of states that are connected only by dummy transitions are the same and, formally, they are in USC conflict. But from the technical point of view, dummies do not correspond to any physical transition made by a circuit, i.e., do not change the 'semantical' state of the system, and one should not interpret such a situation as a coding conflict. However, there are STGs for which an interpretation may be not entirely clear. For example, consider the STG in figure 2 together with a modified state graph, which has been obtained by applying the well-known epsilon-closure procedure described for finite automatons in [16], in order to eliminate all the transitions concerned with firing dummy events. This procedure works on SG_Γ and produces 'essential' edges that are associated with 'transitions' of the form $M \stackrel{\tau^* z\pm}{\longrightarrow} M'$. Such edges lead to a subset of states, which are the result of firing a non-dummy signal transition. Then it removes all τ-labelled transitions and the states which have become unreachable, together with their incoming and outgoing arcs.

This technique treats the markings $\{p_2, p_3\}$ and $\{p_3, p_4\}$ as not equivalent, e.g., because only the former enables a transition labelled by the output signal c. From another point of view, though, these two states are separated only by a dummy transition, and $\{p_3, p_4\}$ may be considered as a hidden intermediate state before t_2 fires at $\{p_2, p_3\}$.

We outline another, *partial order* interpretation of dummies based on the branching processes of STGs. The idea is, given a configuration, to remove all trailing τ-labelled events,

yielding what we call an 'essential' configuration. Other configurations can be seen as 'intermediate', and not corresponding to the 'real' states of the system. Therefore, one may check for coding conflicts involving only essential configuration. Formally, given a configuration of a branching process $\mathcal{C}$, we denote by $\mathit{Strip}(\mathcal{C})$ the minimal configuration containing all its events labelled by a signal transition label. Clearly, $\mathit{Strip}(\mathcal{C})$ is uniquely defined. In figure 2, the net of the STG coincides with its unfolding, and the only 'essential' configurations are $\emptyset$, $\{t_1\}$, $\{t_1, t_2\}$, and $\{t_1, t_3, t_4\}$.

The modified algorithm will consider only the configurations of the form $\mathit{Strip}(\mathcal{C})$ as those which can be in coding conflict. Note that such a semantics has no obvious interpretation on the state graph level, since whether a marking is essential or not may depend not only on the marking itself, but also on the execution path through which it was reached, and on the causal relationships between the transitions involved in this path. Indeed, in the unfolding there may be several configurations with the same final marking M, but some of them have trailing dummies while the others have not. In such a case we say that M exhibits a *backward signal confusion.*

In order to retain only the 'essential' solutions of the system of constraints described in section 3, we can add constraints

$$x(e) \leq \sum_{f \in (e^\bullet)^\bullet} x(f), \quad \text{for all } \tau\text{-labelled events } e \in E \setminus E_{cut} \ ,$$

requiring that if a dummy has fired then at least one of its causal descendants has also fired, and so this dummy is not a trailing one. But this approach may be not completely satisfactory, since the number of constraints increases, and the algorithm still has to consider some of the 'intermediate' configurations just to find out that they are not solutions. A better way is to require that all the maximal events of a configurations be non-dummy. This can be easily achieved by restricting the condition in the header of the **for all** loop of the algorithm in figure 1.

Another problem caused by the dummies is calculating the set of enabled outputs, which is needed for CSC and normalcy checking (note that we need this set to compute the value of the next-state function). Now it is not trivial to compute it, since one can have an STG where it is possible to fire a long sequence of dummies before any signal transition becomes enabled. Furthermore, in the finite prefix of the net unfolding, the correspondent sequence of events may lead beyond the cut-off events.

One solution is to (partially) unfold the STG again, starting from $\mathit{Mark}(\mathcal{C})$ and without generating events beyond any signal event. In order to minimise the number of times this is done, we can first check if the configurations are in USC conflict. Assuming that there are not many CSC conflicts which are not USC conflicts, the performance of the CSC checking algorithm will not suffer much. But for normalcy, the number of times this is done can still be quite large. An alternative solution is to build the prefix beyond the cut-offs, so that any output signal event enabled (possibly, through a sequence of dummies) by any configuration containing no cut-off events can be reached. The algorithm does not have to assign new variables to these added events, it keeps them just for finding the sets of enabled outputs.

What we have described above works for all STGs without markings exhibiting the backward signal confusion. In our future work, we intend to clarify the status of such markings.

It is important to note that proposition 4 holds in the case of STGs containing dummies as well. In such a case, $\mathcal{C}'$ and $\mathcal{C}''$ must have no trailing dummies (otherwise they are not in conflict), and the facts that $\mathit{Strip}(\mathcal{C}) \subseteq \mathcal{C}$ and $\mathit{Code}(\mathcal{C}) = \mathit{Code}(\mathit{Strip}(\mathcal{C}))$ are used in the last stage of the proof.

10 Experimental results

The results of our experiments are summarised in tables 1, 2, and 3. They were measured on a PC with $Pentium^{TM}$ III/500MHz processor and 128M RAM. The meaning of the columns are

Problem	Net			Unfolding			Time, [s]	
	$\|S\|$	$\|T\|$	$\|Sig\|$	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	*Petrify*	*CLP*
LAZYRING	35	32	11	87	66	5	2.02	<0.01
RING	147	127	28	763	498	59	1497.80	0.01
DUP-4PH	133	123	26	144	123	11	35.85	<0.01
DUP-4PH-CSC	135	123	26	146	123	11	34.79	<0.01
DUP-4PH-MTR	109	96	22	117	96	8	25.40	<0.01
DUP-4PH-MTR-CSC	114	105	26	122	105	8	25.26	0.02
DUP-MTR-MOD	129	100	21	199	132	10	221.88	<0.01
DUP-MTR-MOD-UTG	116	165	21	344	218	65	623.23	<0.01
DUP-MTR-MOD-CSC	152	115	27	228	149	13	285.64	<0.01
CF-SYM-A-CSC	85	60	22	1341	720	56	357.43	107.26
CF-SYM-B-CSC	55	32	16	160	71	6	15.27	0.06
CF-SYM-C-CSC	59	36	18	286	137	10	33.24	2.30
CF-SYM-D-CSC	45	28	14	120	54	6	8.57	0.01
CF-ASYM-A-CSC	128	112	34	1808	1234	62	3988.11	2807.48
CF-ASYM-B-CSC	128	112	32	1816	1238	62	6143.50	3280.30

Table 1. Experimental results: real-life STGs.

is 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. We use 'time' to indicate that the test had not stopped after 15 hours. We have not included in the tables the time needed to build complete prefixes, since it did not exceed 0.1sec for all the attempted STGs.

In our experiments we used several groups of benchmarks. The first group consisted of the standard STG benchmarks, which are relatively small STGs coming from the academic practice of control circuit synthesis. We used them for testing the correctness of our implementation, but have not included the timing results in the tables since all these examples were rather trivial. Another group of examples came from the real design practice ([3, 13, 24, 37]). Some STGs, although built by hand, were quite large in size. The results for this group are summarised in table 1.

Two other groups, PipesWeak and PipesArb, contain scalable examples of STGs modelling two and three pipelines weakly synchronised without arbitration (in PipesWeak) and with arbitration (in PipesArb). The former offers the possibility of studying the effect of the optimisation described in section 8 (all STGs in PipesWeak series are marked graphs, and so are free from dynamic conflicts). Figure 3 illustrates these two types of STGs. The versions of these models without the dashed arcs exhibit CSC conflicts; the dashed arrows are indicating the way how CSC conflicts can be resolved by adding extra causality constraints into the specification. Adding them allowed us to test the algorithm on almost identical specifications, which did not contain coding conflicts.

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: recall that it uses just $O(|E|)$ memory besides that needed to store the prefix, which for all the examples shown in the tables means not more than just few kilobytes (in contrast, `Petrify` was reapeatedly 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. If a specification contained a coding conflict, our algorithm in most cases found it very quickly. On the other hand, conflict-free specifications were much

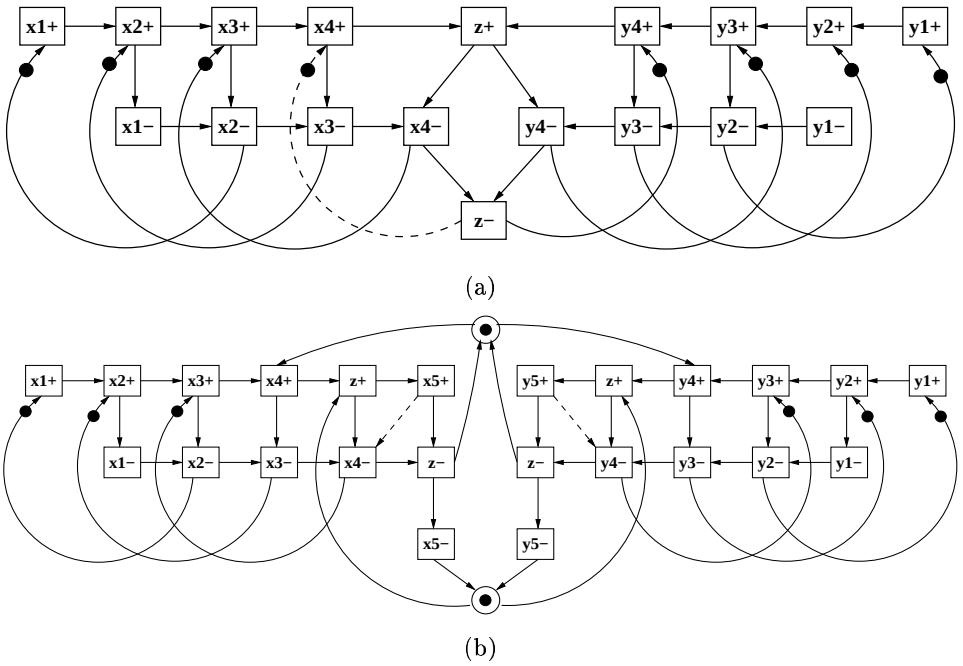

Fig. 3. STG model for two weakly synchronised pipelines (a) without arbitration and (b) with arbitration. Places having one input and one output arc are not shown.

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, which can be much bigger than the number of reachable states. However, the heuristics we described allowed the algorithm to considerably reduce the number of considered configurations.

Conclusions and future work

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.

In our future work, we plan to incorporate more heuristics used by general-purpose solvers in order to reduce the search space. Also, the algorithm admits effective parallelisation ([17]), even for the distributed memory architecture. Additional speedups may be gained by using non-local corresponding configurations, as described in ([15]), for reducing the size of complete prefixes and thus the search space to be explored by our algorithm. Finally, as we already mentioned, we intend to extend the method to STGs with markings exhibiting backward signal confusion.

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 *CDA*.

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).

Problem	Net			Unfolding			Time, [s]	
	$\|S\|$	$\|T\|$	$\|Sig\|$	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	*Petrify*	*CLP*
2PipesWeak(3)	23	14	7	41	23	1	0.55	<0.01
2PipesWeak(6)	47	26	13	119	62	1	12.49	<0.01
2PipesWeak(9)	71	38	19	233	119	1	103.24	<0.01
2PipesWeak(12)	95	50	25	383	194	1	1140.28	0.01
2PipesWeakCsc(3)	24	14	7	38	20	1	0.43	0.01
2PipesWeakCsc(6)	48	26	13	110	56	1	10.73	0.18
2PipesWeakCsc(9)	72	38	19	218	110	1	102.26	16.92
2PipesWeakCsc(12)	96	50	25	362	182	1	4805.52	1410.59
3PipesWeak(3)	34	20	10	63	35	1	2.50	<0.01
3PipesWeak(6)	70	38	19	183	95	1	239.84	0.01
3PipesWeak(9)	106	56	28	357	182	1	4951.52	0.01
3PipesWeak(12)	142	74	37	585	296	1	*time*	0.01
3PipesWeakCsc(3)	36	20	10	57	29	1	1.84	0.01
3PipesWeakCsc(6)	72	38	19	165	83	1	107.53	14.10
3PipesWeakCsc(9)	108	56	28	327	164	1	18281.89	11188.44
3PipesWeakCsc(12)	144	74	37	543	272	1	*time*	*time*

Table 2. Experimental results: marked graphs.

Problem	Net			Unfolding			Time, [s]	
	$\|S\|$	$\|T\|$	$\|Sig\|$	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	*Petrify*	*CLP*
2PipesArb(3)	38	24	11	94	52	2	3.24	<0.01
2PipesArb(6)	62	36	17	202	106	2	37.77	0.47
2PipesArb(9)	86	48	23	346	178	2	883.91	41.11
2PipesArb(12)	110	60	29	520	264	2	5851.06	3818.08
2PipesArbCsc(3)	40	24	11	96	52	2	2.80	0.03
2PipesArbCsc(6)	64	36	17	204	106	2	45.01	2.76
2PipesArbCsc(9)	88	48	23	348	178	2	409.63	231.48
2PipesArbCsc(12)	112	60	29	528	268	2	*time*	19618.11
3PipesArb(3)	56	36	16	161	90	3	18.49	0.02
3PipesArb(6)	92	54	25	341	180	3	1041.89	19.71
3PipesArb(9)	128	72	34	575	297	3	8301.73	14903.17
3PipesArb(12)	164	90	43	863	441	3	*time*	*time*
3PipesArbCsc(3)	59	36	16	164	90	3	15.38	0.51
3PipesArbCsc(6)	95	54	25	344	180	3	449.61	347.86
3PipesArbCsc(9)	131	72	34	578	297	3	16300.04	*time*
3PipesArbCsc(12)	167	90	43	866	441	3	*time*	*time*

Table 3. Experimental results: STGs with arbitration.

References

1. F. Ajili and E. Contejean: Complete Solving of Linear Diophantine Equations and Inequations Without Adding Variables. Proc. of *1st International Conference on principles and practice of Constraint Programming*, Cassis (1995) 1–17.
2. F. Ajili and E. Contejean: Avoiding Slack Variables in the Solving of Linear Diophantine Equations and Inequations. *Theoretical Computer Science* 173 (1997) 183–208.
3. C. Carrion and A. Yakovlev: Design and Evaluation of two Asynchronous Token Ring Adapters. Technical Report CS-TR-562, Department of Computing Science, University of Newcastle (1996).
4. 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).
5. E. Contejean: Solving Linear Diophantine Constraints Incrementally. Proc. of *10th International conference on Logic Programming*, D. S. Warren (Ed.). MIT Press (1993) 532–549.
6. E. Contejean and H. Devie: Solving Systems of Linear Diophantine Equations. Proc. of *3rd Workshop on Unification*, University of Keiserlautern (1989).
7. E. Contejean and H. Devie: An Efficient Incremental Algorithm for Solving Systems of Linear Diophantine Equations. *Information and Computation* 113 (1994) 143–172.
8. 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.
9. J. Cortadella, A. Kondratyev, M. Kishinevsky, L. Lavagno and A. Yakovlev: Complete state encoding based on theory of regions. Proc. of *Second Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, IEEE Computer Society Press (1966) 36–47.
10. J. Engelfriet: Branching processes of Petri Nets. *Acta Informatica* 28 (1991) 575–591.
11. J. Esparza, S. Römer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. Proc. of *TACAS'96: Tools and Algorithms for the Construction and Analysis of Systems*, Margaria T., Steffen B. (Eds.). Springer-Verlag, Lecture Notes in Computer Science 1055 (1996) 87–106.
12. J. Esparza: Model Checking Based on Branching Processes. *Science of Computer Programming* 23 (1994) 151–195.
13. 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 (AINT'2000)*, Yakovlev A., Nouta R. (Eds.). TU Delft, The Netherlands (2000).
14. K. Heljanko: Using Logic Programs with Stable Model Semantics to Solve Deadlock and Reachability Problems for 1-Safe Petri Nets. *Fundamentae Informaticae* 37 (1999) 247–268.
15. K. Heljanko: Minimizing Finite Complete Prefixes. Proc. of *Workshop Concurrency, Specification and Programming 1999 (CS&P'99)*, (1999) 83–95.
16. J. E. Hopcroft and J. D. Ullman: *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley (1979).
17. V. Khomenko and M. Koutny: Deadlock Checking Using Liner Programming and Partial Order Dependencies. Technical Report CS-TR-695, Department of Computing Science, University of Newcastle (2000).
18. V. Khomenko and M. Koutny: Verification of Bounded Petri Nets Using Integer Programming. Technical Report. Technical Report CS-TR-711, Department of Computing Science, University of Newcastle (2000).
19. 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.
20. V. Khomenko and M. Koutny: Verification of Bounded Petri Nets Using Integer Programming. *Formal Methods in System Design* (submitted paper).
21. 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 *International Conference on Application of Concurrency to System Design*, IEEE Computer Society Press (1998) 152–163.
22. A. Kondratyev, J. Cortadella, M. Kishinevsky, E. Pastor, O. Roig and A. Yakovlev: Checking Signal Transition Graph Implementability by symbolic BDD traversal. Proc. of *European Design and Test Conference*, IEEE Computer Society Press (1995) 325–332.
23. S. Krivoy: About Some Methods of Solving and Feasibility Criteria of Linear Diophantine Equations over Natural Numbers Domain (in Russian). *Cybernetics and System Analysis* 4 (1999) 12–36.

24. K. S. Low and A. Yakovlev: Token Ring Arbiters: an exercise in asynchronous logic design with Petri nets. Technical Report CS-TR-537, Department of Computing Science, University of Newcastle (1995).
25. K. L. McMillan: Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits. Proc. of *4th Workshop on Computer Aided Verification*, Springer-Verlag, Lecture Notes in Computer Science 663 (1992) 164–174.
26. K. L. McMillan: *Symbolic Model Checking. An approach to the state explosion problem.* Technical Report CMU-CS-92-131 (1992).
27. S. Melzer: *Verifikation verteilter Systeme mit linearer — und Constraint-Programmierung.* PhD Thesis. Technische Universität München, Utz Verlag (1998).
28. 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.
29. M. A. Peña and J. Cortadella: Combining Process Algebras and Petri Nets for the Specification and Synthesis of Asynchronous Circuits. Proc. of *Second Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, IEEE Computer Society Press (1996) 222–232.
30. A. Pnueli: Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. Proc. of *Advanced School on Current Trends in Concurrency*, Springer-Verlag, Lecture Notes in Computer Science 224 (1994) .
31. A. Semenov: *Verification and Synthesis of Asynchronous Control Circuits Using Petri Net Unfolding.* PhD Thesis, University of Newcastle upon Tyne (1997).
32. A. Semenov, A. Yakovlev, E. Pastor, M. Peña, J. Cortadella and L. Lavagno: Partial order approach to synthesis of speed-independent circuits. Proc. of *Third Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, IEEE Computer Society Press (1997) 254–265.
33. M. Silva, E. Teruel and J. -M. Colom: Linear Algebraic and Linear Programming Techniques for the Analysis of Place/Transition Net Systems. In: *Lectures on Petri Nets I: Basic Models*, Wolfgang Reisig and Grzegorz Rozenberg (Eds.). Springer-Verlag (1998) 309–373.
34. N. Starodoubtsev, S. Bystrov, M. Goncharov, I. Klotchkov and A. Smirnov: Towards Synthesis of Monotonic Asynchronous Circuits from Signal Transition Graphs. Proc. of *Second Int. Conf. on Application of Concurrency to System Design (ICACSD'01)*, IEEE Computer Society Press (2001) 179–188.
35. P. Vanbekbergen, F. Catthoor, G. Goossens and H. De Man: Optimized Synthesis of Asynchronous Control Circuits form Graph-Theoretic Specifications. Proc. of *International Conf. Computer-Aided Design (ICCAD'90)*, IEEE CS Press (1990) 184–187.
36. A. Yakovlev, L. Lavagno and A. Sangiovanni-Vincentelli: A Unified Signal Transition Graph Model for Asynchronous Control Circuit Synthesis. *Formal Methods of System Design* 9(3) (1996) 139–188.
37. A. Yakovlev: Designing Control Logic for Counterflow Pipeline Processor Using Petri nets. *Formal Methods in Systems Design* 12(1) (1998) 39–71.