Form Method Syst Des (2007) 30:143–176
DOI 10.1007/s10703-006-0022-1

Verification of bounded Petri nets using integer programming

Victor Khomenko · Maciej Koutny

Published online: 31 August 2006

Abstract Model checking based on the causal partial order semantics of Petri nets is an approach widely applied to cope with the state space explosion problem. One of the ways to exploit such a semantics is to consider (finite prefixes of) net unfoldings—themselves a class of acyclic Petri nets—which contain enough information, albeit implicit, to reason about the reachable markings of the original Petri nets. In [19], a verification technique for net unfoldings was proposed, in which deadlock detection was reduced to a mixed integer linear programming problem. In this paper, we present a further development of this approach. The essence of the proposed modifications is to transfer the information about causality and conflicts between the events involved in an unfolding, into a relationship between the corresponding integer variables in the system of linear constraints. Moreover, we present some problem-specific optimisation rules, reducing the search space. To solve other verification problems, such as mutual exclusion or marking reachability and coverability, we adopt Contejean and Devie's algorithm for solving systems of linear constraints over the natural numbers domain and refine it, by taking advantage of the specific properties of systems of linear constraints to be solved.

Another contribution of this paper is a method of re-formulating some problems specified in terms of Petri nets as problems defined for their unfoldings. Using this method, we obtain a memory efficient translation of a deadlock detection problem for a safe Petri net into an LP problem. We also propose an on-the-fly deadlock detection method.

Experimental results demonstrate that the resulting algorithms can achieve significant speedups.

Keywords Verification · Petri nets · Integer programming · Net unfoldings · Partial order techniques

V. Khomenko (✉) · M. Koutny
School of Computing Science, University of Newcastle, Newcastle upon Tyne, NE1 7RU, UK
e-mail: victor.khomenko@ncl.ac.uk

M. Koutny
e-mail: maciej.koutny@ncl.ac.uk

1. Introduction

A distinctive characteristic of reactive concurrent systems is that their sets of local states have descriptions which are both short and manageable, and the complexity of their behaviour comes from highly complicated interactions with the external environment rather than from complicated data structures and manipulations thereon. One way of coping with this complexity problem is to use formal methods and, especially, computer aided verification tools implementing model checking [3]—a technique in which the verification of a system is carried out using a finite representation of its state space. The main drawback of model checking is that it suffers from the *state space explosion* problem; that is, even a relatively small system specification can (and often does) yield a very large state space.

To alleviate this problem, a number of methods have been proposed. Among them, a prominent technique is McMillan's (finite prefixes of) Petri net unfoldings [8, 9, 16]. It relies on the partial order view of concurrent computation, and represents system states implicitly, using an acyclic net, called a *prefix*. Often such prefixes are exponentially smaller than the corresponding reachability graphs, especially if the system at hand exhibits a lot of concurrency. The net unfolding technique presented in [16, 19] reduces the memory consumption, but the deadlock checking algorithms proposed there were relatively slow, even for medium-size unfoldings.

In [19], the problem of deadlock checking a Petri net was reduced to a mixed integer linear programming (*MIP*) problem. In this paper, we present a further development of this approach. The essence of the proposed modifications is to transfer the information about causality and conflicts between events involved in an unfolding into a relationship between the corresponding integer variables in the system of linear constraints. We adopt the Contejean and Devie's algorithm (*CDA*), developed in [1, 4], for efficiently solving systems of linear constraints over the domain of natural numbers, and refine it by employing specific properties of the systems of linear constraints to be solved, in model checking aimed at deadlock detection. The results of initial experiments demonstrate that the resulting algorithms can achieve significant speedups.

The paper is organised as follows. In Section 2 we provide basic definitions concerning Petri nets and, in particular, net unfoldings. Section 3 briefly recalls the results presented in [19], where the deadlock checking problem has been reduced to the feasibility test of a system of linear constraints. Section 4 is based on the results developed in [1, 4] and recalls the main aspects of *CDA*. The algorithm we propose in this paper is developed specifically to exploit partial order dependencies between events in the unfolding of a Petri net, combining this idea with *CDA*. It is described in Sections 6 and 7, where we provide theoretical background and implementation details, as well as outlining ways of reducing the number of variables and constraints in the original system presented in [19]. Section 8 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. This method is then used in Section 8.1 to obtain a memory efficient translation of a deadlock detection problem into an LP problem, and in Section 8.3 to deal with other verification problems, such as mutual exclusion, coverability and reachability analysis. Section 9 presents some additional heuristics, which can be incorporated into our algorithm. There we also consider an on-the-fly version of our algorithm, which allows one to verify deadlock-freeness without explicitly generating the system of constraints. Section 10 contains the results of experiments obtained for a number of benchmark examples, and we will finish with some conclusions in Section 11.

2. Basic definitions

In this section, we first present basic definitions concerning Petri nets, and then recall (see also [7–9, 11, 16, 20, 22]) 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* (we will sometimes identify the flow relation with the corresponding characteristic function $(S \times T) \cup (T \times S) \to \{0, 1\}$). 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 ${}^\bullet t \neq \emptyset$, 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 $M' \stackrel{\text{df}}{=} M - {}^\bullet t + t^\bullet$; we denote this by $M[t\rangle M'$. The set of *reachable* markings of Σ is the smallest (w.r.t. $\subset$) set $[M_0\rangle$ containing M_0 and such that if $M \in [M_0\rangle$ and $M[t\rangle M'$ for some $t \in T$ then $M' \in [M_0\rangle$. A marking M is *covered* by marking M' if $M(s) \leq M'(s)$, for all $s \in S$. For a finite sequence $\sigma = t_1 \ldots t_k$ of transitions, we write $M_0[\sigma\rangle M$ if there are markings $M_1, \ldots, M_k$ such that $M_k = M$ and $M_{i-1}[t_i\rangle M_i$, for $i = 1, \ldots, k$.

A marking is *deadlocked* if it does not enable any transitions; the net system Σ is *deadlock-free* if none of its reachable markings is deadlocked. Σ is *k-bounded* if, for every reachable marking M and every place $s \in S$, $M(s) \leq k$, *safe* if it is 1-bounded, and *bounded* if it is k-bounded for some $k \in \mathbb{N}$. The set $[M_0\rangle$ of reachable markings of Σ is finite iff Σ is bounded.

An example of a safe Petri net modelling two dining philosophers is given in Fig. 2(a); it is not deadlock-free since, e.g., the marking $\{s_3, s_4, s_{10}, s_{13}\}$ reached by firing the sequence of transitions $t_1 t_2 t_6 t_8$ is deadlocked.

2.2. Marking equation

Let $\Sigma = (N, M_0)$ be a net system, $S = \{s_1, \ldots, s_m\}$ and $T = \{t_1, \ldots, t_n\}$ be the sets of its places and transitions, respectively, and σ be a finite sequence transitions of Σ such that $M_0[\sigma\rangle M$. By counting the tokens brought to and taken from a place s by the transitions in σ it is possible to calculate $M(s)$ as follows:

$$M(s) = M_0(s) + \sum_{t \in T} F((t, s))\#_t\sigma - \sum_{t \in T} F((s, t))\#_t\sigma,$$

where $\#_t\sigma$ denotes the number of times a transition t occurs in σ, as illustrated in Fig. 1(a). This is a linear equation which holds for every place of Σ. It can be written in the matrix form as follows. We 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 $\mathcal{N} = (\mathcal{N}_{ij})$ such that, for all $i \leq m$ and $j \leq n$, $\mathcal{N}_{ij} \stackrel{\text{df}}{=} F((t_j, s_i)) - F((s_i, t_j))$. The *Parikh vector* of a finite sequence

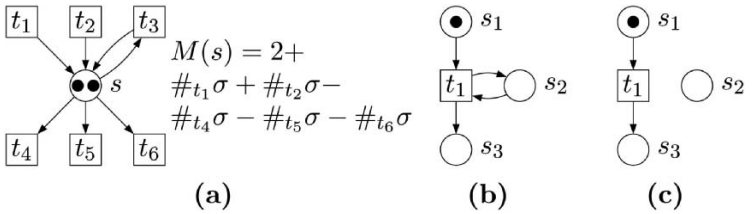


Fig. 1 Marking equation (only one place with its environment and initial marking is shown) (a), and two net systems which have distinct sets of reachable markings but are indistinguishable by the marking equation (b,c). (Note that these net systems have the same incidence matrix and the same initial marking, and so the same set of solutions of the marking equation.)

of transitions σ is a vector $x_\sigma = (x_1, \ldots, x_n)$, where $x_i \stackrel{\text{df}}{=} \#_{t_i}\sigma$, for every $i \leq n$. One can show that if σ is an execution sequence such that $M_0[\sigma\rangle M$ then $M = M_0 + \mathcal{N} \cdot x_\sigma$. This provides a motivation for investigating the feasibility (or solvability) of the following system of equations:

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

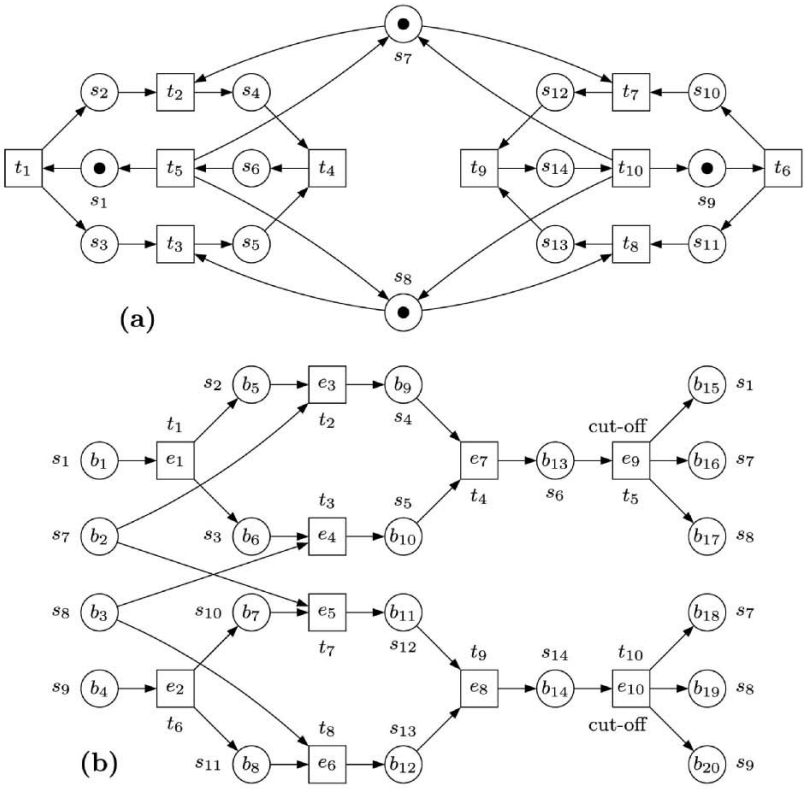
If one fixes the marking M, then the feasibility of the above system is a necessary (but, in general, not sufficient) condition for M to be reachable from M_0.

A vector $x \in \mathbb{N}^n$ is Σ-*compatible* if it is the Parikh vector of some execution sequence of Σ. Each Σ-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.

2.3. Unfolding prefixes

A *finite and complete unfolding prefix* $Pref_\Sigma$ of a Petri net Σ is a finite acyclic net which implicitly represents all the reachable states of Σ together with transitions enabled at those states. Intuitively, it can be obtained through unfolding Σ, by successive firings of transitions, under the following assumptions: (a) for each new firing a fresh transition (called an *event*) is generated; (b) for each newly produced token a fresh place (called a *condition*) is generated. The resulting object is called the *unfolding* of Σ. We will denote by h the function mapping the events and conditions of the unfolding to the corresponding places and transitions of Σ. The unfolding is acyclic, and the precedence relation $\preceq$ on its nodes will be called the *causal order*. Moreover, for two distinct nodes, y and y', of the unfolding: y and y' are *in conflict*, denoted $y \# y'$, if there are distinct events e and e' such that ${}^\bullet e \cap {}^\bullet e' \neq \emptyset$ and $e \preceq y$ and $e' \preceq y'$; y is in *self-conflict* if $y \# y$; and y and y' are *concurrent*, denoted $y \parallel y'$, if neither $y \# y'$ nor $y \preceq y'$ nor $y' \preceq y$.

A *configuration* C is a finite set of events of the unfolding of Σ such that (i) for every $e \in C$, $f \preceq e$ implies $f \in C$ (i.e., C is causally closed), and (ii) for all distinct $e, f \in C$, $\neg(e \# f)$ (i.e., there are no choices between the events of C). Intuitively, a configuration is a partial-order execution, i.e., an execution where the order of firing of some of its events (viz. concurrent ones) is not important. For an event e of the unfolding, $[e] \stackrel{\text{df}}{=} \{f \mid f \preceq e\}$ is called the *local configuration* of e. Moreover, for a configuration C of the unfolding, $Mark(C)$ will



$$
\begin{cases}
b_1 = 1 - e_1 & b_2 = 1 - e_3 - e_5 & b_3 = 1 - e_4 - e_6 & b_4 = 1 - e_2 \\
b_5 = e_1 - e_3 & b_6 = e_1 - e_4 & b_7 = e_2 - e_5 & b_8 = e_2 - e_6 \\
b_9 = e_3 - e_7 & b_{10} = e_4 - e_7 & b_{11} = e_5 - e_8 & b_{12} = e_6 - e_8 \\
b_{13} = e_7 - e_9 & b_{14} = e_8 - e_{10} & b_{15} = e_9 & b_{16} = e_9 \\
b_{17} = e_9 & b_{18} = e_{10} & b_{19} = e_{10} & b_{20} = e_{10} \\
b_1 \le 0 & b_4 \le 0 & b_2 + b_5 \le 1 & b_3 + b_6 \le 1 \quad b_2 + b_7 \le 1 \\
b_3 + b_8 \le 1 & b_9 + b_{10} \le 1 & b_{11} + b_{12} \le 1 & b_{13} \le 0 \quad b_{14} \le 0 \\
e_9 = e_{10} = 0 & (e_1, \ldots, e_{10}) \in \{0,1\}^{10} & (b_1, \ldots, b_{20}) \ge \mathbf{0}
\end{cases}
\tag{c}
$$

$$
\begin{cases}
-e_1 \le -1 & -e_2 \le -1 & e_1 - 2e_3 - e_5 \le 0 & e_1 - 2e_4 - e_6 \le 0 \\
e_2 - e_3 - 2e_5 \le 0 & e_2 - e_4 - 2e_6 \le 0 & e_3 + e_4 - 2e_7 \le 1 & \\
e_5 + e_6 - 2e_8 \le 1 & e_7 - e_9 \le 0 & e_8 - e_{10} \le 0 & \\
e_9 = e_{10} = 0 & (e_1, \ldots, e_{10}) \in \{0,1\}^{10} \text{ is } \mathit{Unf}_\Sigma\text{-compatible}
\end{cases}
\tag{d}
$$

Fig. 2 A Petri net modelling two dining philosophers (a), a finite and complete prefix of its unfolding (b), and the mixed-integer (2) and fully integer (11) linear programming formulations of the deadlock detection problem (c, d)

denote the *final marking of* C, i.e., the marking of Σ reached by the execution $h(e_1)h(e_2)\ldots h(e_k)$, where $e_1, e_2, \ldots, e_k$ is any total ordering of the events of C consistent with $\preceq$.

The unfolding of Σ is infinite whenever Σ has an infinite run; however, if Σ has finitely many reachable states (or, equivalently, Σ is bounded) then the unfolding eventually starts to repeat itself and can be truncated (by identifying a set E_{cut} of *cut-off* events beyond which it is not generated), yielding a finite and complete unfolding prefix $Pref_\Sigma$. Unfolding algorithms declare an event e cut-off if there is a *corresponding configuration* C in the already built part of the prefix containing no cut-off events and satisfying $Mark(C) = Mark([e])$ and $C \lhd [e]$, where $\lhd$ is some well-founded partial order, called an *adequate order*, see [8, 11].

It turns out that prefixes built in this way are *complete*, i.e., (i) every reachable marking M of Σ is represented in such a prefix by means of a configuration C containing no cut-off events and such that $Mark(C) = M$; and (ii) all the firings are preserved, i.e., if a configuration C of $Pref_\Sigma$ containing no cut-off events is such that $Mark(C)$ enables a transition t of Σ then C can be extended by an event e of $Pref_\Sigma$ such that $h(e) = t$ (e may be a cut-off event). Hence, the unfolding is truncated *without loss of information* and can, in principle, be re-constructed from $Pref_\Sigma$. For example, Fig. 2(b) shows a finite and complete prefix of the Petri net in Fig. 2(a); the mapping h is shown as node labels.

Efficient algorithms exist for building finite and complete prefixes [8, 11], which ensure that the number of non-cut-off events in the resulting prefix never exceeds the number of reachable states of Σ. In fact, complete prefixes are often exponentially smaller than the corresponding state graphs, especially for highly concurrent Petri nets, because they represent concurrency directly rather than by multidimensional 'diamonds' as it is done in state graphs. For example, if the original Petri net consists of 100 transitions which can fire once in parallel, the state graph will be a 100-dimensional hypercube with 2^{100} vertices, whereas the complete prefix will coincide with the net itself. The experimental results in [8] demonstrate that high levels of compression can indeed be achieved in practice.

3. Deadlock detection using linear programming

In the rest of this paper we will assume that $Unf_\Sigma \stackrel{\text{df}}{=} (B, E, G, M_{in})$ is the safe net system built from a finite and complete prefix $Pref_\Sigma$ of the unfolding of a bounded net system $\Sigma = (S, T, F, M_0)$, where M_{in} is the canonical initial marking of Unf_Σ which places a single token in each of the minimal (w.r.t. $\preceq$) conditions and no token elsewhere, $E = \{b_1, b_2, \ldots, b_p\}$ and $E = \{e_1, e_2, \ldots, e_q\}$ are respectively the conditions and events of $Pref_\Sigma$, and $\mathcal{C}$ is the $p \times q$ incidence matrix of Unf_Σ.[1] The set of cut-off events of $Pref_\Sigma$ will be denoted by $E_{cut} \subseteq E$.

We now recall the main results from [19]. Since $Pref_\Sigma$ is complete, each reachable deadlocked marking of Σ is represented by a deadlocked marking of Unf_Σ. However, Unf_Σ can have additional deadlocks introduced by truncating the unfolding of Σ. Such deadlocks can be eliminated by prohibiting the cut-off events from occurring. Since for an acyclic Petri net the feasibility of the marking equation is a sufficient condition for a marking to be reachable, the problem of deadlock checking can be reduced to the feasibility test of a system of linear constraints.

[1] We will often identify Unf_Σ and $Pref_\Sigma$, provided that this does not create an ambiguity.

Theorem 1 (Melzer and Römer [19]). Σ *is deadlock-free iff the following system has no solution (in M and x)*:

$$\begin{cases} M = M_{in} + \mathcal{C} \cdot x \\ \sum_{b \in {}^\bullet e} M(b) \leq |{}^\bullet e| - 1 & \text{for all } e \in E \\ x(e) = 0 & \text{for all } e \in E_{cut} \\ M \in \mathbb{N}^p \text{ and } x \in \mathbb{N}^q, \end{cases} \tag{2}$$

where $x(e_i) = x_i$, *for every* $i \leq q$.

In order to decrease the number of integer variables, $M \geq \mathbf{0}$ can be treated as a rational vector, since $x \in \mathbb{N}^q$ and $M = M_{in} + \mathcal{C} \cdot x \geq \mathbf{0}$ always imply that $M \in \mathbb{N}^p$. Moreover, as an event can occur at most once in a given execution sequence of Unf_Σ from the initial marking M_{in}, one can require x to be a binary vector, $x \in \{0,1\}^q$. Figure 2(c) shows the *MIP* formulation (2) of the deadlock detection problem for the unfolding prefix in Fig. 2(b).

To solve the resulting *MIP* problem, [19] used the general-purpose LP-solver CPLEX™ [6], and demonstrated that there are significant performance gains if the number of cut-off events is relatively high, since all variables in x corresponding to cut-off events are set to 0.

We will show in Section 5.1 that it is possible to reduce (2) to a pure integer LP-problem without increasing the total number of integer variables. Moreover, (2) has several problem-specific internal dependencies between variables, and taking them into account may allow one to significantly reduce the number of calculations. Therefore, it turns out to be non-optimal to use general-purpose LP-solvers for this particular problem.

4. Solving systems of linear constraints

In this paper, we will adapt the approach proposed in [1, 4], in order to solve Petri net verification problems which can be reformulated as LP-problems. We start by recalling some basic results.

The original *Contejean and Devie's algorithm (CDA)* [4] solves a system of linear homogeneous equations with arbitrary integer coefficients

$$\begin{cases} a_{11}x_1 + \cdots + a_{1q}x_q = 0 \\ a_{21}x_1 + \cdots + a_{2q}x_q = 0 \\ \quad \vdots \qquad\qquad \vdots \quad \vdots \\ a_{p1}x_1 + \cdots + a_{pq}x_q = 0, \end{cases} \tag{3}$$

or $\mathcal{A} \cdot x = \mathbf{0}$, where $x \in \mathbb{N}^q$ and $\mathcal{A} \stackrel{\text{df}}{=} (a_{ij})$. For every $1 \leq j \leq q$, let

$$\varepsilon_j \stackrel{\text{df}}{=} (\underbrace{0, \ldots, 0}_{j-1 \text{ times}}, 1, 0, \ldots, 0)$$

be the j-th vector in the canonical basis CB of $\mathbb{N}^q$. Moreover, for every $x \in \mathbb{N}^q$, let $a(x) \in \mathbb{N}^p$ be a vector defined by

$$a(x) \stackrel{\text{df}}{=} \begin{pmatrix} a_{11}x_1 + \cdots + a_{1q}x_q \\ a_{21}x_1 + \cdots + a_{2q}x_q \\ \vdots \qquad\qquad \vdots \\ a_{p1}x_1 + \cdots + a_{pq}x_q \end{pmatrix} = \sum_{i=1}^{q} x_i \cdot a(\varepsilon_i), \tag{4}$$

where $a(\varepsilon_j)$—the j-th column vector of the matrix $\mathcal{A}$—is called the *j-th basic default vector.*

The set $\mathcal{S}$ of all solutions of (3) can be represented by a finite basis $\mathcal{B}$ which is the minimal (w.r.t. $\subset$) subset of $\mathcal{S}$ such that every solution is an $\mathbb{N}$-linear combination of the solutions in $\mathcal{B}$. It can be shown that $\mathcal{B}$ comprises all solutions in $\mathcal{S}$ different from the trivial one, $x = \mathbf{0}$, which are minimal with respect to the $\leq$ ordering on $\mathbb{N}^q$ ($x \leq x'$ if $x_i \leq x'_i$, for all $1 \leq i \leq q$; moreover, $x < x'$ if $x \leq x'$ and $x \neq x'$).

The representation (4) suggests that any solution of (3) can be seen as a multiset of default vectors whose sum is $\mathbf{0}$. Choosing an arbitrary order among these vectors amounts to constructing a sequence of default vectors starting from, and returning to, the origin of $\mathbb{Z}^p$. *CDA* constructs such a sequence step by step: starting from the empty sequence, new default vectors are added until a solution is found, or no minimal solution can be obtained. However, different sequences of default vectors may correspond to the same solution (up to permutation of vectors). To eliminate some of the redundant sequences, a restriction for choosing the next default vector is used.

Branching Condition 1. A vector $x \in \mathbb{N}^q$ (corresponding to a sequence of default vectors) such that $a(x) \neq \mathbf{0}$ can be incremented by 1 on its j-th component provided that $a(x + \varepsilon_j) = a(x) + a(\varepsilon_j)$ lies in the half-space containing $\mathbf{0}$ and delimited by the affine hyperplane perpendicular to the vector $a(x)$ at its extremity when originating from $\mathbf{0}$ (see Fig. 3).

This reflects a view that $a(x)$ should not become too large, hence adding $a(\varepsilon_j)$ to $a(x)$ should yield a vector $a(x + \varepsilon_j) = a(x) + a(\varepsilon_j)$ 'returning to the origin'. Formally, this restriction can be expressed by saying that given $x = (x_1, \ldots, x_q)$,

$$\text{increment by 1 an } x_j \text{ satisfying } a(x) \odot a(\varepsilon_j) < 0, \tag{5}$$

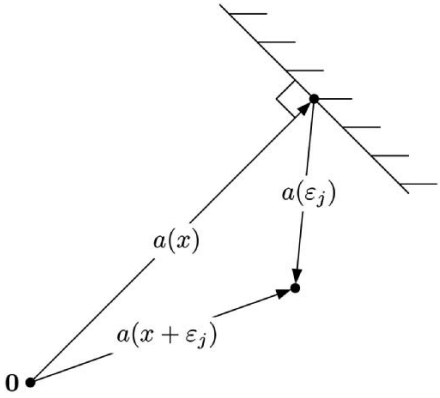


Fig. 3 Geometric interpretation of the branching condition in *CDA*

search breadth-first a directed acyclic graph rooted at $\varepsilon_1, \ldots, \varepsilon_q$
if a node y is equal to, or greater than, an already found
solution of $\mathcal{A} \cdot x = \mathbf{0}$
then y is a terminal node
otherwise construct the sons of y by computing $y + \varepsilon_j$
for each $j \leq q$ satisfying $a(y) \odot a(\varepsilon_j) < 0$

Fig. 4 An outline of *CDA* (breadth-first version)

where $\odot$ denotes the scalar product of two vectors. This reduces the search space without losing any minimal solution, since every sequence of default vectors which corresponds to a solution can be rearranged into a sequence satisfying (5).

Theorem 2 (Contejean and Devie [4]). *The following hold for CDA shown in Fig. 4*:

1. *Every minimal solution of the system* (3) *is found. (completeness)*
2. *Every solution found by CDA is minimal. (soundness)*
3. *The algorithm always terminates. (termination)*

Figure 5(a) illustrates the process of solving the homogeneous system of linear equations

$$\begin{cases} -x_1 + x_2 + 2x_3 - 3x_4 = 0 \\ -x_1 + 3x_2 - 2x_3 - x_4 = 0, \end{cases}$$

considered in [15]. The example shows redundancies, as some vectors were computed more than once. This can be remedied by using *frozen components*, defined as follows. Assume that there is a total ordering $\prec_x$ on the sons of each node[2] x of the search graph constructed by *CDA*.

Frozen Components 1. If $x+\varepsilon_i$ and $x+\varepsilon_j$ are two distinct sons of a node x such that $x + \varepsilon_i \prec_x x + \varepsilon_j$ then the j-th component is *frozen* in the sub-graph rooted at $x+\varepsilon_i$ and cannot be incremented even if the condition in (5) is satisfied.

The modified algorithm is still complete [4], and builds a forest which is a sub-graph of the original search graph. By defining[3] the ordering $\prec_x$ as $x + \varepsilon_i \prec_x x + \varepsilon_j \Leftrightarrow i < j$ we obtain, for the system in the above example, the graph shown in Fig. 5(b) [15].

The ordered version of *CDA* can easily handle bounds imposed on variables:

– $x' \leq x$. Then, instead of starting with the vectors $\varepsilon_1, \ldots, \varepsilon_q$, the algorithm starts with x'. The rest of the operation remains the same, but the minimal elements of the set $\mathcal{S}' = \{x \mid \mathcal{A} \cdot x = \mathbf{0} \wedge x' \leq x\}$ do not give all the solutions of

$$\begin{cases} \mathcal{A} \cdot x = \mathbf{0} \\ x' \leq x. \end{cases}$$

[2] Including the virtual node $\mathbf{0}$.

[3] The ordering $\prec_x$ may be defined in other ways as well [4].

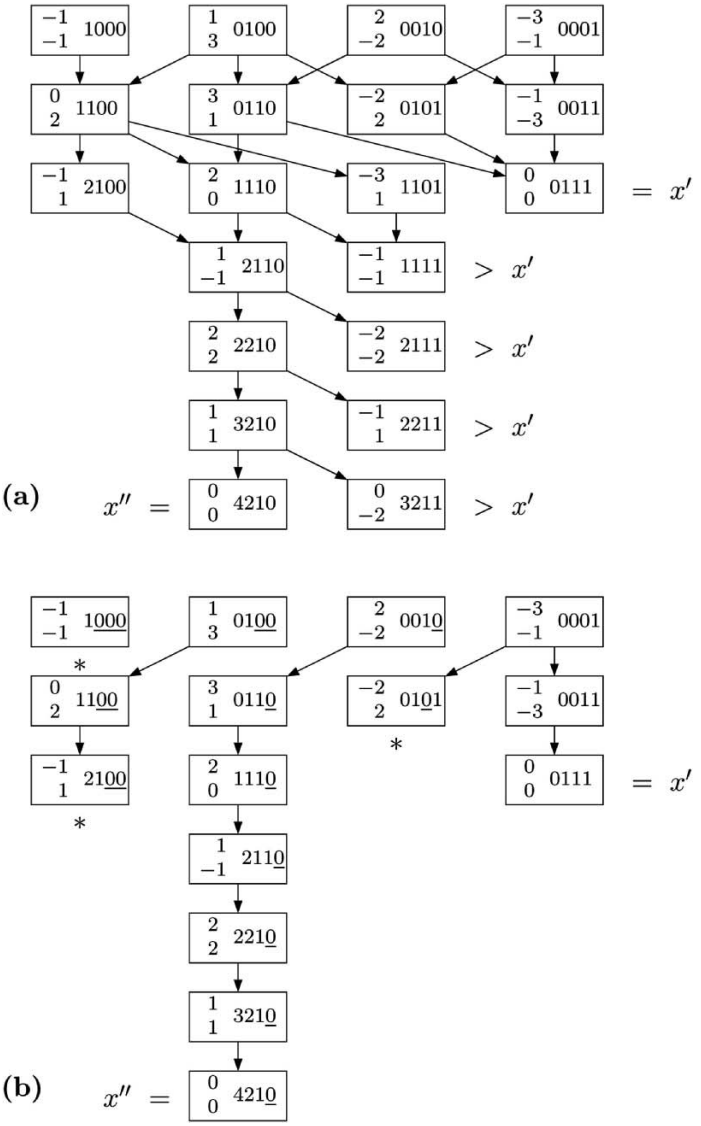


Fig. 5 (a) Search graph constructed by *CDA* in Fig. 4; inside each box, the current value of $a(x)$ is represented by a column on the left, and is followed by the current value of x; note that $x' = (0, 1, 1, 1)$ and $x'' = (4, 2, 1, 0)$ are two minimal solutions. (b) Search graph constructed by the ordered version of *CDA*; frozen components are underlined, and the *s indicate the nodes which cannot be developed due to condition (5) and the frozen components rule

However, any solution of the above system can be represented as a sum of a minimal element of $\mathcal{S}'$ and an $\mathbb{N}$-linear combination of minimal solutions of the original system.

– $x \leq x''$ where $x'' \in (\mathbb{N} \cup \{\infty\})^q$. Then the algorithm works in the standard way except that the j-th component of a vector becomes frozen as soon as it reaches the j-th component of x''.
– $x' \leq x \leq x''$. Then a combination of two previous techniques is used.

With the above extensions, *CDA* allows one to solve non-homogeneous diophantine systems

$$\begin{cases} a_{11}x_1 + \cdots + a_{1q}x_q = d_1 \\ a_{21}x_1 + \cdots + a_{2q}x_q = d_2 \\ \quad\vdots \qquad\qquad\quad \vdots \quad \vdots \\ a_{p1}x_1 + \cdots + a_{pq}x_q = d_p. \end{cases} \tag{6}$$

By introducing a new variable, x_0, one can transform (6) into a homogeneous system

$$\begin{cases} -d_1x_0 + a_{11}x_1 + \cdots + a_{1q}x_q = 0 \\ -d_2x_0 + a_{21}x_1 + \cdots + a_{2q}x_q = 0 \\ \quad\vdots \qquad\quad \vdots \qquad\qquad\quad \vdots \quad \vdots \\ -d_px_0 + a_{p1}x_1 + \cdots + a_{pq}x_q = 0. \end{cases}$$

Let $\mathcal{B}_k$ $(k = 0, 1)$ be the set of all minimal solutions $x = (x_0, x_1, \ldots, x_q)$ of this system with $x_0 = k$. Then any solution of (6) can be represented as

$$x = y + \sum_{z \in \mathcal{B}_0} c_z\, z,$$

where $y \in \mathcal{B}_1$ and each c_z belongs to $\mathbb{N}$. Thus, to solve (6), it suffices to add just one variable which becomes frozen as soon as it reaches the value 1.

The task of solving a system of linear inequalities

$$\begin{cases} a_{11}x_1 + \cdots + a_{1q}x_q \le d_1 \\ a_{21}x_1 + \cdots + a_{2q}x_q \le d_2 \\ \quad\vdots \qquad\qquad\quad \vdots \quad \vdots \\ a_{p1}x_1 + \cdots + a_{pq}x_q \le d_p \end{cases} \tag{7}$$

is more complicated. In general, not all the solutions of (7) can be represented as $\mathbb{N}$-linear combinations of minimal solutions, even if the system of inequalities is homogeneous. As an example, [1] considers the inequality $x - y \le 0$. Its only non-trivial minimal solution is (0,1), which is not enough to generate the set of all solutions, $\{(n, n+m) \mid n, m \in \mathbb{N}\}$. To generate the whole set one needs also to take a non-minimal solution $(1, 1) > (0, 1)$.

The standard linear programming approach is to reduce (7) to a system of equations

$$\begin{cases} a_{11}x_1 + \cdots + a_{1q}x_q + y_1 \qquad\quad = d_1 \\ a_{21}x_1 + \cdots + a_{2q}x_q + \quad y_2 \qquad\quad = d_2 \\ \quad\vdots \qquad\qquad\quad \vdots \qquad\quad \ddots \quad \vdots \\ a_{p1}x_1 + \cdots + a_{pq}x_q + \qquad\quad y_p = d_p \end{cases}$$

by introducing *slack* variables $y_i \in \mathbb{N}$, but this transformation increases the number of variables from q to $q + p$. Consequently, as the computation time can grow exponentially in the

number of variables, such an approach is not efficient. Moreover, slack variables may assume arbitrary values in $\mathbb{N}$, even if all the variables in the original problem were binary as in (2); as a result, the search space can grow very rapidly.

Another approach is to deal with the inequalities (7) directly. It was developed in [1], where *CDA* has been generalised to solve homogeneous systems of equations and inequalities. The approach uses the notion of a *non-decomposable* solution, i.e., one which cannot be represented as an $\mathbb{N}$-linear combination of other solutions; one can see that the non-decomposable solutions form a basis of the set of all the solutions. For a system of linear constraints $\mathcal{A} \cdot x = \mathbf{0} \wedge \mathcal{B} \cdot x \leq \mathbf{0}$, the branching condition (5) is modified in the following way.

Branching Condition 2. Given a vector $x = (x_1, \ldots, x_q)$, increment by 1 an element x_j for which there exist $y_1, \ldots, y_p$ such that the vector $(x_1, \ldots, x_q, y_1, \ldots, y_p)$ can be incremented on its j-th component according to (5) applied to the system $\mathcal{A} \cdot x = \mathbf{0} \wedge \mathcal{B} \cdot x + y = \mathbf{0}$, where p is the number of rows in $\mathcal{B}$ and $y = (y_1, \ldots, y_p)$.

As shown in [1], this condition can be expressed as

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

where $\mathcal{B}_i$ is the i-th row of $\mathcal{B}$. To ensure the termination in the general case, [1] added one more condition, but if all the variables are bounded (as in our case) then such a condition is unnecessary.

5. Integer programming verification algorithm

In this section we start by turning the deadlock detection problem—one of the fundamental verification problems for Petri nets—into a pure integer problem. We then 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 develop an extension of *CDA* aimed at combining these dependencies with the original algorithm.

5.1. Reduction to a pure integer problem

The *MIP* problem described in Section 3 can be reduced to a pure integer one, by substituting the expression for M given by the marking equation into the other constraints. Each equation in $M = M_{in} + \mathcal{C} \cdot x$ has the form

$$M(b) = M_{in}(b) + \sum_{f \in {}^\bullet b} x(f) - \sum_{f \in b^\bullet} x(f),$$

where $b \in B$. Substituting these into (2) results in the following system

$$\begin{cases} \sum_{b \in {}^\bullet e} \left(\sum_{f \in b^\bullet} x(f) - \sum_{f \in {}^\bullet b} x(f) \right) \leq |{}^\bullet e| - 1 - \sum_{b \in {}^\bullet e} M_{in}(b) & \text{for all } e \in E \\ M_{in} + \mathcal{C} \cdot x \geq 0 & \text{for all } b \in B \\ x \in \{0,1\}^q \text{ and } x(e) = 0 & \text{for all } e \in E_{cut}. \end{cases} \tag{9}$$

Usually, each inequality in (9) contains relatively few variables, so it does make sense to use the sparse-matrix representation of this system. For efficiency reasons, inequalities can be first generated without paying attention to possible repetitions of the same variable in its left-hand side, and then sorted and transformed into the normal form. But one should be careful when choosing the sorting algorithm: the sequence of monomials obtained after generating the inequalities is often nearly sorted, and `QuickSort` performs rather poorly, i.e., in quadratic time. Our early experiments showed that in this case the process of sorting monomials can be much more time consuming than the process of solving the system; it is therefore better to use a sorting algorithm with $O(n \log n)$ worst case execution time. In our implementation, we obtained satisfactory results with `HeapSort`, which has an additional advantage that it does not require auxiliary arrays.

As (9) is a pure integer problem, the usual integer programming algorithms are in principle directly applicable. However, since the number of variables is usually large even for moderate sized net systems, a further refinement is needed.

5.2. Partial-order dependencies between variables

In [19], Unf_Σ is used only for building a system of constraints, and the latter is then passed to an LP-solver without any additional information. Yet, during the solving of the system, one may use dependencies between variables implied by the causal order on events, which can easily be derived from Unf_Σ. For example, if we set $x(e) = 1$ then each $x(f)$ such that $f \preceq e$ must be equal to 1, and each $x(g)$ such that $g \# e$, must be equal to 0. Similarly, if we set $x(e) = 0$ then no event f satisfying $e \preceq f$ can be executed in the same run, and so $x(f)$ must be equal to 0. Hence, it is sufficient to restrict the search space to vectors satisfying these constraints.

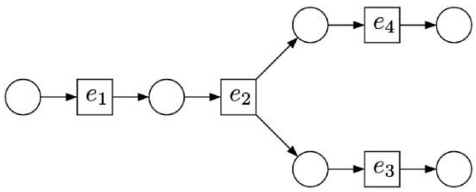
These observations can be formalised by considering Unf_Σ-compatible vectors (see Section 2 for the definition), which in this case correspond to the configurations of $Pref_\Sigma$, and the following result provides a basis for such an approach.

Theorem 3. *A vector $x \in \{0,1\}^q$ is Unf_Σ-compatible iff for all distinct events $e, f \in E$ such that $x(e) = 1$, we have*:

$$f \preceq e \Rightarrow x(f) = 1 \quad \text{and} \quad f \# e \Rightarrow x(f) = 0. \tag{10}$$

Proof: Straightforward. We just point out that Unf_Σ-compatible vectors are binary, since each event in the unfolding of Σ can occur at most once in an execution sequence, and vectors satisfying (10) correspond to configurations of $Pref_\Sigma$. □

Fig. 6 An unfolding prefix



Corollary 1. *For each reachable marking M of Σ, there exists an execution sequence of Unf_Σ leading to a marking representing M, whose Parikh vector x satisfies* (10)*, and for every $e \in E_{cut}$, $x(e) = 0$.*

Proof: Since the prefix $Pref_\Sigma$ used to build Unf_Σ was complete, each reachable marking M of Σ is represented in Unf_Σ by a marking M' which can be reached from M_{in} through an execution sequence σ without cut-off events. Theorem 3 implies that the Parikh vector of σ satisfies (10). □

In view of this result, it is sufficient for a deadlock detection algorithm to check only Unf_Σ-compatible vectors whose components corresponding to cut-off events are equal to zero. This can be done by freezing all $x(e)$ such that $e \in E_{cut}$ at the beginning of the algorithm and constructing the *minimal Unf_Σ-compatible closure* (see below) of the current vector in each step of the algorithm.

5.3. Compatible closures

An Unf_Σ-compatible vector $y \in \{0, 1\}^q$ is an *Unf_Σ-compatible closure* of a vector $x \in \{0, 1\}^q$ if $x \leq y$. Moreover, y is the *minimal Unf_Σ*-compatible closure of x, denoted by $MCC(x)$, if it is minimal (w.r.t. $\leq$) among all Unf_Σ-compatible closures of x. Note that $MCC(x)$ is undefined for some x's, but whenever it is defined then, due to Theorem 4 below, it is unambiguous.

As an example, consider the prefix shown in Fig. 6, and let $x = (1, 0, 1, 0)$. Then $y = (1, 1, 1, 0)$ and $z = (1, 1, 1, 1)$ are Unf_Σ-compatible closures of x, and $MCC(x) = y$.

Theorem 4. *A vector $x \in \{0, 1\}^q$ has an Unf_Σ-compatible closure iff for all $e, f \in E$, $x(e) = x(f) = 1$ implies $\neg(e \# f)$. If x has an Unf_Σ-compatible closure then its minimal Unf_Σ-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 $MCC(x)$.*

Proof: Straightforward. We just point out that to build the minimal Unf_Σ-compatible closure of x, when it does exist, it is enough to set to 1 all the components $x(f)$ for which there is e such that $f \preceq e$ and $x(e) = 1$, i.e., to 'downclose' the set of events corresponding to x, producing a configuration. □

From the implementation point of view, it may happen that a vector x has an Unf_Σ-compatible closure according to Theorem 4, but it cannot be computed because some of the zero components of x to be set to 1 have been frozen during the search process (see Section 4). In such a case, the algorithm should behave as if such a closure could not be built.

5.4. Removing redundant constraints

One can see that the inequalities in the middle of (9) are not essential for an algorithm checking only Unf_Σ-compatible vectors. Indeed, such a vector x corresponds to some execution sequence of Unf_Σ, and so the marking $M = M_{in} + \mathcal{C} \cdot x$ is guaranteed to be non-negative. Consequently, these inequalities may be left out without adding any Unf_Σ-compatible solution. The reduced system

$$\begin{cases} \displaystyle\sum_{b \in {}^\bullet e} \left(\sum_{f \in {}^\bullet b} x(f) - \sum_{f \in b^\bullet} x(f) \right) \le |{}^\bullet e| - 1 - \sum_{b \in {}^\bullet e} M_{in}(b) & \text{for all } e \in E \\ x \in \{0,1\}^q \text{ and } x(e) = 0 & \text{for all } e \in E_{cut} \\ x \text{ is } Unf_\Sigma\text{-compatible} \end{cases} \tag{11}$$

is the one which will be treated in the rest of this paper. Figure 2(d) shows the formulation (11) of the deadlock detection problem for the unfolding prefix in Fig. 2(b).

5.5. Extending CDA (intuition)

Each step of *CDA* can be seen as moving from a point $a(x)$ along a default vector $a(\varepsilon_j)$ such that $a(x) \odot a(\varepsilon_j) < 0$, which is interpreted as 'returning to the origin' (see Fig. 3). However, for an algorithm checking Unf_Σ-compatible vectors only, each step consists in moving along a vector which may be represented as a sum of *several* default vectors, and this branching condition is no longer valid. Indeed, let us consider the same ordering as in Fig. 6, and the equation

$$a(x) = x_1 + 5x_2 - 3x_3 - 3x_4 = 0$$

(which has a solution $x = (1, 1, 1, 1)$) with an initial constraint $x_1 = 1$. The algorithm starts from the vector $x = (1, 0, 0, 0)$, and the sequence of steps should begin from either ε_2 or $\varepsilon_2 + \varepsilon_3$ or $\varepsilon_2 + \varepsilon_4$. But $a(x) \odot a(\varepsilon_2) = 5 \not< 0$, $a(x) \odot a(\varepsilon_2 + \varepsilon_3) = 2 \not< 0$ and $a(x) \odot a(\varepsilon_2 + \varepsilon_4) = 2 \not< 0$, i.e., one cannot choose a vector to make the first step!

A possible solution is to interpret each step $\varepsilon_{i_1} + \cdots + \varepsilon_{i_k}$ as a sequence of smaller steps, $\varepsilon_{i_1}, \ldots, \varepsilon_{i_k}$, where the algorithm chooses only the first element ε_{i_1} for which $a(\varepsilon_{i_1})$ does return to the origin, and then builds the minimal Unf_Σ-compatible closure $x + \varepsilon_{i_1} + \cdots + \varepsilon_{i_k}$ of $x + \varepsilon_{i_1}$ without worrying where the vector $\varepsilon_{i_1} + \cdots + \varepsilon_{i_k}$ actually leads (if there is no Unf_Σ-compatible closure of $x + \varepsilon_{i_1}$ then ε_{i_1} cannot be chosen). This means that the algorithm uses the condition $a(x) \odot a(\varepsilon_{i_1}) < 0$ which coincides with the original *CDA*'s branching condition, though it is moving along a possibly different vector. The geometric interpretation of this new branching condition is shown in Fig. 7. We will now cast the above idea in a formal setting.

6. Developing an extension of CDA

In this section, we will obtain a general result extending that in [4]. This result will later be applied to unfolding-based model checking, but it is also of independent interest.

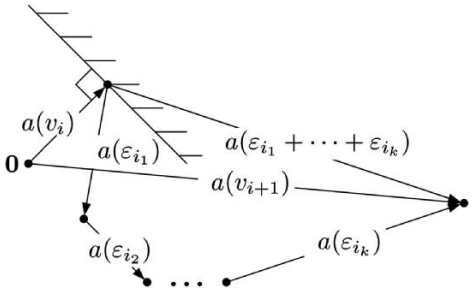


Fig. 7 Geometric interpretation of the new branching condition: $a(\varepsilon_{i_1})$ is 'returning to the origin', although $a(\varepsilon_{i_1}+\cdots+\varepsilon_{i_k})$ may not posses this property; here $v_{i+1} = v_i + \varepsilon_{i_1} + \cdots + \varepsilon_{i_k}$ is the minimal Unf_Σ-compatible closure of $v_i + \varepsilon_{i_1}$

6.1. Branching condition

Consider the following homogeneous system of linear constraints:

$$\begin{cases} \mathcal{A}\cdot x = \mathbf{0} \\ \mathcal{B}\cdot x \leq \mathbf{0} \\ x \in D \stackrel{\text{df}}{=} D_1 \times \cdots \times D_q, \end{cases} \tag{12}$$

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 $i \leq q$. Below we assume that $\mathbf{0} \notin D$.[4]

Let $\xi : D \to D$ be a partial function[5] 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 (12) 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:

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

The aim is to develop an algorithm searching for all ξ-minimal solutions and, in what follows, we present an extension of *CDA* achieving this. First, we introduce a new branching condition.

Branching Condition 3. A vector $x \in codom$ which is not a solution of (12) can be extended to $\xi(x+\varepsilon_j)$ if $x + \varepsilon_j \in dom$ and

$$(\mathcal{A}\cdot x) \odot (\mathcal{A}\cdot \varepsilon_j) + \sum_{i=1}^{m} \min \left\{ \begin{array}{c} \max\{0, \mathcal{B}_i \odot x\}(\mathcal{B}_i \odot \varepsilon_j), \\ (\mathcal{B}_i \odot x)(\mathcal{B}_i \odot \varepsilon_j) \end{array} \right\} < 0, \tag{14}$$

where m is the number of rows in $\mathcal{B}$, and $\mathcal{B}_i$ is the i-th row of $\mathcal{B}$.

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 $A \subseteq X \times CB \times X$ is a set

[4] 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.

[5] Later we will take ξ to be the *MCC* function to apply the developed technique to unfolding-based model checking.

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 (12), if $\varepsilon_j \in CB$ satisfies $x + \varepsilon_j \in dom$ and (14) then $(x, \varepsilon_j, \xi(x + \varepsilon_j)) \in A$. Directly from the definitions we obtain

Proposition 1. *G_ξ is finite and acyclic.*

Proof: From the first part of (13) it follows that $x < y$, for every $(x, \varepsilon_j, y) \in A$. Thus a directed path in G_ξ can have at most $|D|$ vertices. The result follows from this and $|X| \leq |D| < \infty$. □

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 $z \leq y$.*

Proof: (Adapted from [1]) We have $y - x = \sum_{j \in \mathcal{J}} \varepsilon_j$, for some non-empty *multiset* $\mathcal{J}$. Suppose that the desired arc does not exist. We observe that, for every $j \in \mathcal{J}$, by the second part of (13), $x + \varepsilon_j \in dom$ and $\xi(x + \varepsilon_j) \leq y$. Thus, for all $j \in \mathcal{J}$,

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

and after summing these inequalities for all $j \in \mathcal{J}$, we obtain

$$(\mathcal{A} \cdot x) \odot (\mathcal{A} \cdot (y - x)) + \sum_{j \in \mathcal{J}} \sum_{i=1}^{m} \min \left\{ \begin{array}{c} \max\{0, \mathcal{B}_i \odot x\}(\mathcal{B}_i \odot \varepsilon_j), \\ (\mathcal{B}_i \odot x)(\mathcal{B}_i \odot \varepsilon_j) \end{array} \right\} \geq 0. \tag{15}$$

Let $\mathcal{I}$ and $\mathcal{K}$ be the sets of all $i \leq m$ such that $\mathcal{B}_i \odot x > 0$ and $\mathcal{B}_i \odot x \leq 0$, respectively. Since $\mathcal{A} \cdot y = \mathbf{0}$,

$$\sum_{j \in \mathcal{J}} \sum_{i \in \mathcal{I}} (\mathcal{B}_i \odot x)(\mathcal{B}_i \odot \varepsilon_j) \geq ||\mathcal{A} \cdot x||^2 - \sum_{j \in \mathcal{J}} \sum_{i \in \mathcal{K}} \min\{(\mathcal{B}_i \odot x)(\mathcal{B}_i \odot \varepsilon_j), 0\} \geq 0.$$

We are now going to show that $\mathcal{I} = \emptyset$. Indeed, by the last inequality,

$$\sum_{j \in \mathcal{J}} \sum_{i \in \mathcal{I}} (\mathcal{B}_i \odot x)(\mathcal{B}_i \odot \varepsilon_j) = \sum_{i \in \mathcal{I}} (\mathcal{B}_i \odot x)(\mathcal{B}_i \odot (y - x)) \geq 0.$$

This, and the fact that for all $i \in \mathcal{I}$, $(\mathcal{B}_i \odot x)(\mathcal{B}_i \odot y) \leq 0$ (which follows from $\mathcal{B} \cdot y \leq \mathbf{0}$ and the definition of $\mathcal{I}$), yields

$$0 \geq \sum_{i \in \mathcal{I}} (\mathcal{B}_i \odot x)(\mathcal{B}_i \odot y) \geq \sum_{i \in \mathcal{I}} (\mathcal{B}_i \odot x)^2 \geq 0.$$

Hence $\mathcal{B}_i \odot x = 0$, for all $i \in \mathcal{I}$. This, however, means that $\mathcal{I} = \emptyset$. As a result, $\mathcal{B} \cdot x \leq \mathbf{0}$.

From $\mathcal{I} = \emptyset$ it further follows that $\max\{\mathcal{B}_i \odot x, 0\} = 0$, for all $i \leq m$, which together with (15) and $\mathcal{A} \cdot y = \mathbf{0}$ leads to

$$\sum_{j \in \mathcal{J}} \sum_{i=1}^{m} \min\{(\mathcal{B}_i \odot x)(\mathcal{B}_i \odot \varepsilon_j), 0\} \geq ||\mathcal{A} \cdot x||^2 \geq 0.$$

Thus, since $\min\{(\mathcal{B}_i \odot x)(\mathcal{B}_i \odot \varepsilon_j), 0\} \leq 0$, for every $i \leq m$, we obtain that $\mathcal{A} \cdot x = \mathbf{0}$. Hence $x \in codom$ is a solution of (12) satisfying $x < y$, contradicting $y \in min_\xi$. □

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

Proof: Let $x \in min_\xi$. We first observe that $x_{root} \leq x$ which follows from the second part of (13). Hence $x \in X$, by Propositions 1 and 2. □

6.2. Frozen components

Although Proposition 1 and Corollary 2 imply that G_ξ could be used to solve the problem at hand,[6] it may contain a large number of redundant paths. We will now adapt the frozen components method of [1, 4] 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 2. 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 $(x, \varepsilon_i, \xi(x + \varepsilon_i))$.

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 CB such that

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

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

Proof: We first observe that the orderings associated with the vertices of G_ξ induce, for every vertex x, a total order $\ll_x$ on all the directed paths leading from x_{root} to x in such a way that, $\sigma \ll_x \sigma'$ iff $\sigma = \sigma_1(y, \varepsilon_i, z)\sigma_2, \sigma' = \sigma_1(y, \varepsilon_j, z')\sigma_3$ and $\varepsilon_i \prec_y \varepsilon_j$ (note that since

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

G_ξ is acyclic, a directed path leading from x_{root} to x cannot be a prefix of another directed path from x_{root} to x).

Suppose that T_ξ is not a tree. Then there are two different non-frozen directed paths, $\sigma \ll_x \sigma'$, leading from x_{root} to some node $x \neq x_{root}$. We can represent them as $\sigma = \sigma_1(y, \varepsilon_i, z)\sigma_2$ and $\sigma' = \sigma_1(y, \varepsilon_j, z')\sigma_3$, where $\varepsilon_i \neq \varepsilon_j$. Then $\varepsilon_j \in Froz_\sigma(\alpha)$, for every arc in $(y, \varepsilon_i, z)\sigma_2$. Moreover, by the first part of (13), $\varepsilon_j \leq x - y$ and so σ is not non-frozen, a contradiction.

Suppose now that $x \in min_\xi$. Since G_ξ is finite, and there is at least one directed path from x_{root} to x, there is a unique directed path $\sigma = \alpha_1 \ldots \alpha_k$ from x_{root} to x which is maximal w.r.t. $\ll_x$. Suppose that such a σ is not non-frozen. Then there are $m \in \{1, \ldots, k\}$ and $\alpha \in A$ such that $\alpha_m = (y, \varepsilon_j, \xi(y + \varepsilon_j))$, $\alpha = (y, \varepsilon_i, \xi(y + \varepsilon_i))$, $\varepsilon_j \prec_y \varepsilon_i$ and $\varepsilon_i \in Supp(x - y)$. By the second part of (13), $\xi(y + \varepsilon_i) \leq x$. Hence, by Propositions 1 and 2, there is a directed path $\sigma' = \alpha\alpha'_1 \ldots \alpha'_l$ from $\xi(y + \varepsilon_i)$ to x. Thus $\sigma'' = \alpha_1 \ldots \alpha_{m-1}\sigma'$ is a directed path in G_ξ such that $\sigma \ll_x \sigma''$, contradicting the choice of σ. Hence x is a vertex of T_ξ. □

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 will given in the form of an additional function *froz*.

Frozen Components 3. We assume that, for every arc α of T_ξ, $froz(\alpha)$ is a subset of CB 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.

Theorem 6. *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.*

Proof: Follows directly from the definitions and (13). □

To summarise, Branching Condition 3 and Frozen Components 2 and 3 define search trees which can be traversed[7] to find all ξ-minimal solution of (12) in a finite number of steps (as G_ξ is finite, see Proposition 1).

6.3. Non-homogeneous systems

The developed approach can be applied to deal with a non-homogeneous system of linear constraints

$$\begin{cases} \mathcal{A} \cdot x = a \\ \mathcal{B} \cdot x \leq b \\ x \in D, \end{cases} \tag{16}$$

where we *do not* assume that $\mathbf{0} \notin D$, and all the notions and assumptions relating to ξ are as those for (12).

The problem of finding all ξ-minimal solutions of (16) can be reduced to an instance of the problem considered earlier in this section. To this end, we introduce an auxiliary variable

[7] Using, e.g., depth-first search as the breadth-first search would be inefficient due to the need to record frozen components.

z and two matrices, $\mathcal{A}' \stackrel{\text{df}}{=} (\mathcal{A}, -a)$ and $\mathcal{B}' \stackrel{\text{df}}{=} (\mathcal{B}, -b)$. Then (16) can be rewritten as

$$\begin{cases} \mathcal{A}' \cdot (x, z) = \mathbf{0} \\ \mathcal{B}' \cdot (x, z) \leq \mathbf{0} \\ (x, z) \in D' \stackrel{\text{df}}{=} D \times \{1\}. \end{cases} \tag{17}$$

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 (12) (note that $\mathbf{0} \notin D'$). We now observe that x is a ξ-minimal solution of (16) iff $(x, 1)$ is a ξ'-minimal solution of (17). As a result, one can render the branching condition derived for (17), directly in terms of (16).

Branching Condition 4. A vector $x \in codom$ which is not a solution of (16) can be extended to $\xi(x + \varepsilon_j)$ if $x + \varepsilon_j \in dom$ and

$$(\mathcal{A} \cdot x - a) \odot (\mathcal{A} \cdot \varepsilon_j) + \sum_{i=1}^{m} r_i < 0\,, \tag{18}$$

where, for every $1 \leq i \leq m$,

$$r_i \stackrel{\text{df}}{=} \begin{cases} 0 & \text{if } \mathcal{B}_i \odot x \leq b_i \text{ and } \mathcal{B}_i \odot \varepsilon_j \leq 0 \\ (\mathcal{B}_i \odot x - b_i)(\mathcal{B}_i \odot \varepsilon_j) & \text{otherwise.} \end{cases}$$

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 (12) 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 developed earlier in this section still hold, in particular, Theorems 5 and 6.

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. ◇

6.4. Optimisations

Various heuristics used by general purpose integer programming 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 [4]).

Moreover, if the algorithm, having *fixed* some variables,[8] finds out that some of the inequalities have become infeasible, then it may prune the current branch of the search tree. Alternatively, 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 [12], some simple heuristics of this sort, applicable to arbitrary systems of linear constraints, were described).

After fixing the value of a variable, the ξ-function is computed on the resulting vector. As new variables can become fixed during this process, the above tests can be applied iteratively. (If the ξ-function increases some of the frozen components then the current subtree of the search tree contains no ξ-minimal 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:

$$\begin{gathered} x \in dom_{opt} \Rightarrow x \leq opt(x) \\ x \leq y \in min_{\xi} \Rightarrow x \in dom_{opt} \ \wedge \ opt(x) \leq y. \end{gathered} \tag{19}$$

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 (13) and (19):

$$\begin{gathered} x \in dom_o \Rightarrow x \leq \xi_o(x) \\ x \leq y \in min_{\xi} \Rightarrow x \in dom_o \ \wedge \ \xi_o(x) \leq y. \end{gathered} \tag{20}$$

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

Proof: Suppose that $x \in min_{\xi}$. Then, by $x \leq x$ and (20), we have $x \leq \xi_o(x) \leq x$. Hence $\xi_o(x) = x$ and so $x \in codom_o$. If $x \notin min_{\xi_o}$, then there is $y \in min_{\xi_o}$ such that $y < x$. Hence, since $codom_o \subseteq codom$, we obtained a contradiction with $x \in min_{\xi}$.

Suppose that $y \in min_{\xi_o}$. If $y \notin min_{\xi}$ then, by $codom_o \subseteq codom$, there is $z \in min_{\xi}$ such that $z < y$. By the first part of the proof, $z \in min_{\xi_o}$, contradicting $y \in min_{\xi}$. □

From Proposition 3 and (20) it follows that the counterpart of (13) 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 more efficient than using S_{ξ}. As for the frozen components given by the function $froz_o$, it must satisfy the condition in Frozen Components 3.

In order to avoid calculations related to redundant constraints, one 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 [13].

[8] x_i is fixed if it is equal to the highest value in D_i, or if ε_i has been frozen.

[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 $y \geq x$.

7. Applying the method for Unf_Σ-compatible vectors

We will now apply the theory developed in the previous section to check only Unf_Σ-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:

- $D_i \stackrel{\text{df}}{=} \{0\}$ if $e_i \in E_{cut}$, and $D_i \stackrel{\text{df}}{=} \{0, 1\}$ otherwise.
- dom is the set of all vectors of D having an Unf_Σ-compatible closure, and $\xi(x) \stackrel{\text{df}}{=} MCC(x)$.
- For an arc $\alpha = (x, \varepsilon_j, y)$, $froz(\alpha) \stackrel{\text{df}}{=} \{\varepsilon_i \mid \exists \varepsilon_k \in Supp(y) : e_k \# e_i\}$.

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_Σ-compatible solutions. Further optimisations can also be incorporated, as described at the end of previous section.

Retrieving a solution

What we often want to see as a solution is an execution sequence of the original net system, rather then a configuration of its unfolding. To derive such a sequence, it is enough to topologically sort the constructed configuration according to the causal order on the set of the events, and replace the events by their labels in the constructed sequence. An observation one can make is that the existing unfolding algorithms [8, 11] 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, $e_i \preceq e_j$ implies $i \leq j$. Therefore, if the natural numbering of the components of x, $x_i = x(e_i)$, is used then one can avoid sorting the events and find a sequence of transitions in a straightforward way.

Shortest trail

Finding a shortest path leading, e.g., to a deadlock can facilitate the debugging of the system modelled by a Petri net. In such a case, one has to solve an optimisation problem with the same system of constraints, and $\mathcal{L}(x) = x_1 + \cdots + x_q$ as the cost function to be minimised.

The described approach can easily be adopted for this task. Indeed, it is enough for the algorithm not to stop after the first solution has been found, but to keep the current optimal solution together with the corresponding value of the function $\mathcal{L}$. As this function is non-decreasing, one can prune a branch of the search tree as soon as the 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 the completeness of the algorithm is not affected in the sense that a ξ-minimal solution minimising $\mathcal{L}$ is computed when 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' branch. Since the orderings $\prec_x$ used by the algorithm are arbitrary, one can exploit the information about the value of $\mathcal{L}$ on the successors of x, and check those with smaller values first (see, e.g., the $\prec_{\|\cdot\|}$ ordering in [4]). Such an algorithm can be seen as a version of the 'branch and

bound' method which considers only Unf_Σ-compatible vectors and uses frozen components and branching condition to reduce the search space.

8. Extended reachability analysis

The algorithm described in Section 7 is applicable to any system of linear constraints which are supposed to be solved in Unf_Σ-compatible vectors. The theory of verifying *co-linear* properties using unfoldings was developed in [18]. It can easily be generalised to arbitrary reachability properties, although solving (very large) non-linear systems obtained in this case is usually a hard task for general-purpose solvers. An algorithm checking only Unf_Σ-compatible vectors can do this more efficiently. Indeed, the only reason why the algorithm in Section 7 accepts only systems of linear constraints is that in order to reduce the search space it employs the branching condition (18). 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 in [18], generalised to deal with non-linear constraints. In addition, we use the ideas from Section 5 to re-formulate the resulting LP problem as a problem based on Unf_Σ-compatible vectors and reduce the number of variables and constraints in the system.

Let $\mathcal{P}$ be a property specified for the markings of the original net system Σ. One can transform it into a corresponding property $\mathcal{P}'$ specified for Unf_Σ-compatible vectors in such a way that if there exists a reachable marking M in Σ for which $\mathcal{P}$ holds then $\mathcal{P}'$ holds for some Unf_Σ-compatible vector, and vice versa. Indeed, let M be a reachable marking of Σ, and M' be a corresponding marking in Unf_Σ. Then $M(s)$ can be calculated as

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

where the marking $M'(b)$ of a place b in Unf_Σ can be found from the marking equation

$$M'(b) = M_{in}(b) + \sum_{f \in {}^\bullet b} x(f) - \sum_{f \in b^\bullet} x(f) \ .$$

Therefore,

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

and $\mathcal{P}'$ can be rendered as a predicate specified for Unf_Σ-compatible vectors. And, moreover, if $\mathcal{P}$ is initially expressed as a system of linear constraints then $\mathcal{P}'$ will possess this property as well.

8.1. Deadlock checking in safe case

By applying the technique described in the previous section, one can generate a system of constraints different from (11) for deadlock checking safe Petri nets (a similar idea was used in [18] to obtain a translation of this problem into a *MIP* problem). To begin with, the

following condition states for safe Petri nets that no transition is enabled:

$$\sum_{s\in{}^\bullet t} M(s) \le |{}^\bullet t| - 1 \quad \text{for all } t \text{ in } T \setminus T_d, \tag{21}$$

where T_d is the set of transitions which are dead in Σ. For example, referring to the Petri net and its unfolding in Fig. 2(a) and (b), the constraint for transition t_2 of has the form $M(s_2)+M(s_7)\le 1$, where $M(s_2) = e_1 - e_3$ (since $h^{-1}(s_2) = \{b_5\}$) and $M(s_7) = 1 - e_3 - e_5 + e_9 + e_{10}$ (since $h^{-1}(s_7) = \{b_2, b_{16}, b_{18}\}$).

Rendering (21) in terms of the finite prefix yields the following system of linear constraints (all non-dead transitions needed for constructing this system can easily be found, as we have a finite and complete prefix):

$$\begin{cases} \displaystyle\sum_{s\in{}^\bullet t}\sum_{b\in h^{-1}(s)}\left(M_{in}(b) + \sum_{f\in{}^\bullet b} x(f) - \sum_{f\in b^\bullet} x(f)\right) \le |{}^\bullet t| - 1 & \text{for all } t \in T\setminus T_d \\ x \in \{0,1\}^q \text{ and } x(e) = 0 & \text{for all } e \in E_{cut} \\ x \text{ is } Unf_\Sigma\text{-compatible .} & \end{cases} \tag{22}$$

In particular, referring to the example in Fig. 2(a) and (b), the constraint for transition t_2 becomes $1 + e_1 - 2e_3 - e_5 + e_9 + e_{10} \le 1$, or, equivalently, $e_1 - 2e_3 - e_5 + e_9 + e_{10} \le 0$.

We now need to keep in memory at most $|T|$ constraints rather than $|E|$ as in the previous method. Though the constraints are now longer, the overall size of the whole system (in terms of the number of monomials) is often much smaller.

Note that this method is in some sense more general then the one described in Section 3. In the latter, the cut-off events played an essential role in separating real deadlocks from the false ones, introduced by truncating the unfolding. But the notion of a cut-off event is very specific to the existing algorithms used for prefix generation. Consequently, one can imagine an algorithm for generating prefixes, using different principles of cutting unfoldings (to illustrate this issue, [8] considers a prefix shown in Fig. 8). The approach proposed in this section will work for a prefix generated by such a hypothetical algorithm as well.

To apply this approach to non-safe net systems, one can use the following constraints instead of (21):

$$\sum_{s\in{}^\bullet t} sg(M(s)) \le |{}^\bullet t| - 1 \quad \text{for all } t \text{ in } T \setminus T_d \ ,$$

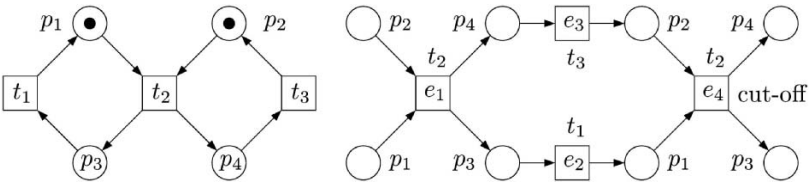


Fig. 8 A Petri net and its finite and complete prefix generated by the algorithm described in [8]; although neither e_2 nor e_3 can be marked as a cut-off event, all the reachable markings are still represented after removing the cut-off event e_4

where $sg(0) \stackrel{\text{df}}{=} 0$ and $sg(n) \stackrel{\text{df}}{=} 1$ for every $n > 0$; or, alternatively,

$$\prod_{s \in {}^\bullet t} M(s) = 0 \quad \text{for all } t \text{ in } T \setminus T_d \ .$$

Although the resulting system is non-linear, it can be dealt with by the algorithm in Section 7 with the branching condition (18) switched off.

8.2. Terminal markings

Some reactive systems can have states corresponding to a proper termination, which are considered to be different from deadlocks, though they may enable no transition. For example, PEP [2] works with a class of labelled nets, called *boxes* [14], which are essentially safe Petri nets with distinguished disjoint sets of *entry* and *exit* places, respectively denoted by $\mathcal{I}$ and $\mathcal{O}$. The proper terminal marking for a box is defined as one that puts a token in each of the exit places and no token elsewhere. Such a false deadlock can be eliminated from the set of solutions by adding a new constraint, which holds for all but terminal markings. As the relevant property $\mathcal{P}$ one can take

$$\sum_{s \in \mathcal{O}} M(s) - \sum_{s \in S \setminus \mathcal{O}} M(s) \leq |\mathcal{O}| - 1,$$

and, using the approach described earlier in this section, render this constraint in terms of Unf_Σ and add it to (11) or (22).

One could slightly relax the notion of a terminal marking of a box, allowing dead tokens on internal (i.e., different from the entry and exit) places. Such a situation can be handled in a similar way using the constraint

$$\sum_{s \in \mathcal{O}} M(s) - \sum_{s \in \mathcal{I}} M(s) \leq |\mathcal{O}| - 1.$$

8.3. Other verification problems

In this section we consider checking mutual exclusion of places, marking reachability and coverability. Since all these properties are linear (or co-linear), they can be verified using the approach proposed in [18]. We refine the technique proposed there, by checking only Unf_Σ-compatible vectors.

Mutual exclusion

Two places, s and s', of a net system Σ are *mutually exclusive* if for any reachable marking M, at least one of them is empty, or, in other words, $M(s) \geq 1$ and $M(s') \geq 1$ cannot hold simultaneously. Using the technique described earlier in this section, one can state that a necessary and sufficient condition for s and s' to be mutually exclusive is the infeasibility of

the following system of linear constraints:

$$\begin{cases} \displaystyle\sum_{b\in h^{-1}(s)}\left(\sum_{e\in{}^\bullet b}x(e)-\sum_{e\in b^\bullet}x(e)\right)\geq 1-\sum_{b\in h^{-1}(s)}M_{in}(b) \\ \displaystyle\sum_{b\in h^{-1}(s')}\left(\sum_{e\in{}^\bullet b}x(e)-\sum_{e\in b^\bullet}x(e)\right)\geq 1-\sum_{b\in h^{-1}(s')}M_{in}(b) \\ x\in\{0,1\}^q \text{ and } x(e)=0 \text{ for all } e\in E_{cut} \\ x \text{ is } Unf_\Sigma\text{-compatible.} \end{cases}$$

In the safe case, one can check the pairwise mutual exclusion of more than two places simultaneously, and still remain within the domain of linear constraints. Indeed, let $S'\subseteq S$ be a set of places whose mutual exclusion should be checked. Then $\sum_{s\in S'}M(s)\geq 2$ must be not satisfied by any reachable marking M of Σ, and so the corresponding necessary and sufficient condition is the infeasibility of the following system:

$$\begin{cases} \displaystyle\sum_{s\in S'}\sum_{b\in h^{-1}(s)}\left(\sum_{e\in{}^\bullet b}x(e)-\sum_{e\in b^\bullet}x(e)\right)\geq 2-\sum_{s\in S'}\sum_{b\in h^{-1}(s)}M_{in}(b) \\ x\in\{0,1\}^q \text{ and } x(e)=0 \text{ for all } e\in E_{cut} \\ x \text{ is } Unf_\Sigma\text{-compatible .} \end{cases}$$

Reachability and coverability
Since it is clear how the standard reachability and coverability problems can be expressed in terms of extended reachability, we give the translation directly. A marking M of Σ is reachable (coverable) iff the following system of linear constraints is feasible:

$$\begin{cases} \displaystyle\sum_{b\in h^{-1}(s)}\left(\sum_{f\in{}^\bullet b}x(f)-\sum_{f\in b^\bullet}x(f)\right)\overset{(\geq)}{=}M(s)-\sum_{b\in h^{-1}(s)}M_{in}(b) & \text{for all } s\in S \\ x\in\{0,1\}^q \text{ and } x(e)=0 & \text{for all } e\in E_{cut} \\ x \text{ is } Unf_\Sigma\text{-compatible.} \end{cases}$$

This system can be simplified even further: for the coverability problem one can leave out the constraints for which $M(s)=0$ as they always hold, and for the reachability problem one can replace all such constraints by their sum.

In the safe case, M should be a safe marking, i.e., $M(s)\in\{0,1\}$ for all $s\in S$ (otherwise it is neither reachable, nor coverable). Therefore, one can add up the other constraints, reducing the system to a single constraint in the case of the coverability problem, and to a system of two constraints for the reachability problem. Moreover, we can replace $\geq$ by $=$ when checking coverability.

Remark 2. One might be tempted to use the following additional heuristic for the problems described above: *increment only those x(e), for which the transition h(e) produces a token on a place in M*. Technically, this can be done by using a problem-specific branching condition. But in some cases the branching condition (18) already allows one to choose only such $x(e)$

to be incremented! Let us show that this holds, e.g., for coverability analysis in the safe case. Each constraint has the form

$$\sum_{b\in h^{-1}(s)}\left(\sum_{f\in b^\bullet} x(f)-\sum_{f\in{}^\bullet b} x(f)\right)\le \sum_{b\in h^{-1}(s)} M_{in}(b)-1 \quad \text{for all } s\in M\ ,$$

where M is the marking we want to cover. If (18) holds for some ε_j then $r_i<0$ for some i, i.e., $(\mathcal{B}_i\odot x-b_i)(\mathcal{B}_i\odot\varepsilon_j)<0$. One can see that $\mathcal{B}_i\odot x\ge b_i$ always holds for safe nets, and so $\mathcal{B}_i\odot\varepsilon_j<0$, i.e., $\mathcal{B}_{ij}$ is negative. But this means that x_j is $x(e)$ for some event $e\in{}^\bullet(h^{-1}(M))$. Note that the above argument can easily be modified for the cases when the constraints are added up and/or $\le$ is replaced by $=$. ◇

9. Further optimisation for deadlock detection

The deadlock detection problem (11), has a very special structure, which can further be exploited. In particular, the maximal value of the left hand side of the inequality

$$\sum_{b\in{}^\bullet e}\left(M_{in}(b)+\sum_{f\in{}^\bullet b} x(f)-\sum_{f\in b^\bullet} x(f)\right)\le |{}^\bullet e|-1$$

is $|{}^\bullet e|$, even if we allow x to be non-Unf_Σ-compatible. Therefore, the i-th inequality in (11) is falsified iff all the variables from Pos_i are equal to 1, and all the variables from Neg_i are equal to 0, where Pos_i and Neg_i denote the sets of the variables with respectively positive and negative coefficients. This means that one can mark the i-th inequality as redundant as soon as any of the variables from Pos_i becomes frozen at 0, or if any of the variables from Neg_i is set to 1. In addition to this simple redundancy test, one can apply on each step an infeasibility test for each non-redundant inequality of (11).

If for the inequality all the variables from Pos_i are set to 1, and all the variables from Neg_i are frozen at 0, then this inequality (and, thus, the whole system) cannot be satisfied, and the algorithm may stop developing the current branch of the search tree. Apart from this, if all but one variable from Pos_i are set to 1, and all the variables from Neg_i are frozen at 0, then the only way to prevent a contradiction is to freeze at 0 the remaining variable from Pos_i. And, similarly, if all the variables in Pos_i are set to 1, and all but one variable in Neg_i are frozen at 0, then we may deterministically set the remaining variable to 1. In both cases, the constraint becomes redundant. Notice that these rules can be justified by choosing appropriate *opt* and $froz_o$ functions (see Section 6.4).

The above heuristics turned out to be much more effective than the general ones described in [12] (see the results in Section 10 and in the corresponding part of [12]).

9.1. The safe case

Unfortunately, the above heuristics do not work for (22), where the inequalities are more complex. But one still can derive some problem-specific optimisation rules.

The maximal value of the left-hand side of the inequality for a transition t on any Unf_Σ-compatible vector x is bounded by $|{}^\bullet t|$, since (22) was obtained from (21), and $M(s)\le 1$ for

safe net systems. Therefore, if for some inequality, say the i-th one, the value of its left-hand side is $|{}^\bullet t|$, then we can state the following:

- If all the variables from Neg_i are frozen at 0, then this inequality (and, thus, the whole system) can never be satisfied. Hence the algorithm may stop developing the current branch of the search tree.
- If all the variables in Pos_i are set to 1, and all but one variable in Neg_i are frozen, then we may deterministically set the remaining non-fixed variable to 1. After this the constraint becomes redundant.

Moreover, if the value of the left-hand side is $|{}^\bullet t| - 1$, and all the variables from Neg_i are frozen at 0, then the only way to prevent a contradiction is to freeze all the non-fixed variables from Pos_i. After this the constraint becomes redundant.

Again, the correctness of these heuristics can be justified by choosing appropriate *opt* and $froz_o$ functions (see Section 6.4).

The problem-specific redundancy tests we obtained for (22) are relatively complex, and not as effective as the test described above for (11). The reason is that the inequalities of (22) do not become redundant as often as those of (11), and we used the general min/max-tests developed in [12]. In Section 10, we will discuss how this new method compares with other deadlock detection algorithms.

9.2. On-the-fly deadlock detection

Our experimental results demonstrated that the algorithm outlined in Section 7 is usually fast, but the treated systems of constraints can be very large, even if the sparse matrix representation is used (see Table 3 in Section 10). This, in turn, can lead to page swapping when checking large unfoldings. Therefore, it is clearly desirable to find a way to reduce the memory consumption, provided that this results in an increase of the running time only by a small factor.

We now observe that the structure of the constraints in (11) is rather simple, and that they can be generated 'on-the-fly', when they are needed. Indeed, the algorithm refers to the system of constraints when checking whether the system is satisfied, when computing the branching condition, and when applying the optimisation rules described in the beginning of this section. All these can be efficiently done on-the-fly, without explicitly generating the system of constraints, by exploring the sets ${}^\bullet({}^\bullet e)$ and $({}^\bullet e)^\bullet$ for all $e \in E$. An observation one can make here is that for any event $e \in E$, ${}^\bullet({}^\bullet e) \cap ({}^\bullet e)^\bullet = \emptyset$ (since $e' \in {}^\bullet({}^\bullet e) \cap ({}^\bullet e)^\bullet$ means that simultaneously e' is a causal predecessor of e and either $e' = e$ or $e' \# e$). Therefore, the positive and negative coefficients for each constraint in (11) can be effectively separated, and this can be exploited by the algorithm.

The on-the-fly approach can, in fact, be applied to other verification problems considered in this paper. However, the resulting gains would not be so significant, as the corresponding systems of constraints are usually of moderate size.

10. Experimental results

The results of our deadlock checking[10] experiments are summarised in Tables 1–3, where we use 'time' to indicate that the test had not stopped after 15 hours, 'mem' to indicate that

[10] Though in Section 8.3 we consider other model checking problems, some of the existing tools we use for comparison support only deadlock checking; moreover, the system of constraints for the other described problems tend to be smaller and easier to solve than those for deadlock detection.

Table 1 All the examples except sdl_arg_d and STACK(n) are deadlock-free

	Unfolding			Time [s]					
							PO		
Problem	$\|B\|$	$\|E\|$	$\|E_{cut}\|$	*McM*	*MIP*	*SM*	*std*	*ext*	*o-t-fly*
buf100	10101	5051	1	0.01	24577	0.17	0.01	0.02	0.01
mutual	887	479	79	4.42	70	0.03	<0.01	0.01	<0.01
ab_gesc	3326	1200	511	33.93	260	0.16	0.02	0.04	0.02
sdl_arg	644	199	10	0.04	20	0.01	<0.01	0.01	<0.01
sdl_arg_d	657	223	7	0.01	25	0.03	<0.01	<0.01	<0.01
PEPRW(2)	498	147	53	0.02	1	0.01	<0.01	<0.01	<0.01
PEPRW(3)	4668	1281	637	22.14	*time*	0.13	0.01	0.02	0.01
PEPRW(4)	51040	13513	7841	*mem*	–	1.47	0.33	0.97	0.76
SEM(2)	61	32	5	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
SEM(3)	165	86	17	<0.01	1	<0.01	<0.01	<0.01	<0.01
SEM(4)	417	216	49	0.09	5	0.01	<0.01	<0.01	<0.01
SEM(5)	1013	522	129	2.30	81	0.03	<0.01	0.01	0.01
SEM(6)	2393	1228	321	37.33	*time*	0.10	0.02	0.05	0.03
SEM(7)	5533	2830	769	531.50	—	0.29	0.11	0.23	0.12
SEM(8)	12577	6416	1793	*time*	–	0.78	0.49	1.38	0.54
SEM(9)	28197	14354	4097	–	–	2.28	2.26	9.42	2.69
SEM(10)	62505	31764	9217	–	–	7.30	10.74	56.10	13.65
PEPEL(1)	518	287	9	0.10	27	0.02	<0.01	<0.01	<0.01
PEPEL(2)	29413	15366	1796	*mem*	*time*	2.91	1.85	10.42	3.06
STACK(3)	320	174	26	<0.01	3	0.01	<0.01	<0.01	<0.01
STACK(4)	968	525	80	0.08	79	0.02	< 0.01	< 0.01	< 0.01
STACK(5)	2912	1578	242	4.28	2408	0.07	< 0.01	< 0.01	< 0.01
STACK(6)	8744	4737	728	145.01	*time*	0.24	0.01	0.01	0.01
STACK(7)	26240	14214	2186	*mem*	–	0.81	0.05	0.05	0.06
STACK(8)	78728	42645	6560	–	–	2.82	0.14	0.23	0.20
STACK(9)	236192	127938	19682	–	–	10.40	0.41	0.83	0.61

the test terminated because of memory overflow, and 'inst' to indicate that the test gave an incorrect result or terminated because of numerical instability. The results in Tables 1–3 were measured on a PC with *Pentium*^{TM} III/500 MHz processor and 128M RAM. The following benchmarks have been attempted:

- PEP examples:

  buf100 – buffer with 2^{100} states
  mutual – mutual exclusion algorithm (by R.Walter)
  ab_gesc – alternating bit protocol
  sdl_arg – automatic request protocol
  sdl_arg_d – automatic request protocol (with a deadlock)
  PEPRW(n) – readers-writers with n readers
  SEM(n) – semaphore with n processes
  PEPEL(n) – n elevators
  STACK(n) – depth n stack with test for fullness

- Examples used in [19]:

  DPD(n) – dining philosophers (dictionary version)

DPH(n) – dining philosophers (host version)
ELEV(n) – n elevators
FURN(n) – remote furnace program
RING(n) – token ring mutual exclusion protocol
RW(n) – database with n readers and writers
DME(n) – distributed mutual exclusion asynchronous circuit with n DME-cells
SYNC(n) – readers/writers synchronisation

The first set of examples was taken from the standard PEP distribution (some of them were scaled). The *DME* series of examples is due to K. McMillan, and the *SYNC* examples

Table 2 All the examples except ELEV(n) are deadlock-free

	Unfolding			Time [s]					
							PO		
Problem	$\|B\|$	$\|E\|$	$\|E_{\text{cut}}\|$	*McM*	*MIP*	*SM*	*std*	*ext*	*o-t-fly*
DPD(4)	594	296	81	0.34	8	0.01	<0.01	<0.01	<0.01
DPD(5)	1582	790	211	21.32	87	0.05	0.01	0.01	0.01
DPD(6)	3786	1892	499	544.42	711	0.17	0.05	0.05	0.05
DPD(7)	8630	4314	1129	11637	8610	0.55	0.26	0.18	0.21
DPH(4)	680	336	117	0.49	8	0.01	<0.01	<0.01	<0.01
DPH(5)	2712	1351	547	66.58	290	0.08	0.02	0.02	0.02
DPH(6)	14474	7231	3377	*time*	31062	0.76	0.32	0.55	0.34
DPH(7)	81358	40672	21427	–	*time*	8.12	4.55	19.15	7.02
ELEV(1)	296	157	59	0.01	<0.01	0.01	<0.01	<0.01	<0.01
ELEV(2)	1562	827	331	0.59	10	0.03	<0.01	<0.01	<0.01
ELEV(3)	7398	3895	1629	83.41	438	0.16	0.04	0.01	0.02
ELEV(4)	32354	16935	7337	*mem*	11962	0.80	0.23	0.08	0.16
FURN(1)	535	326	189	0.15	2	0.01	<0.01	<0.01	<0.01
FURN(2)	5139	3111	1990	235.74	676	0.11	0.03	0.06	0.03
FURN(3)	34505	20770	13837	*mem*	129938	1.00	0.40	2.07	0.40
RING(3)	97	47	11	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
RING(5)	339	167	37	0.05	8	0.01	<0.01	<0.01	<0.01
RING(7)	813	403	79	0.93	181	0.02	<0.01	0.01	<0.01
RING(9)	1599	795	137	12.99	2506	0.07	0.01	0.06	0.01
RW(6)	806	397	327	0.14	2	0.01	<0.01	<0.01	<0.01
RW(9)	9272	4627	4106	166.70	*inst*	0.17	0.03	0.01	0.06
RW(12)	98378	49177	45069	*mem*	*time*	2.70	1.28	0.26	4.85
DME(2)	487	122	4	0.02	9	0.01	<0.01	<0.01	<0.01
DME(3)	1210	321	9	0.42	354	0.04	<0.01	<0.01	<0.01
DME(4)	2381	652	16	3.54	30	0.09	0.01	0.01	0.01
DME(5)	4096	1145	25	26.99	123	0.23	0.04	0.05	0.04
DME(6)	6451	1830	36	184.19	256	0.57	0.10	0.13	0.11
DME(7)	9542	2737	49	929	*inst*	1.39	0.22	0.35	0.25
DME(8)	13465	3896	64	3802	*inst*	3.11	0.55	0.98	0.63
DME(9)	18316	5337	81	12560	–	7.91	1.25	2.59	1.48
DME(10)	24191	7090	100	36800	–	21.38	2.90	6.82	3.51
DME(11)	31186	9185	121	96600	–	59.84	6.68	17.42	8.12
SYNC(2)	4007	2162	490	274.79	4403	0.11	0.05	0.14	0.06
SYNC(3)	29132	15974	5381	*mem*	*time*	1.52	1.46	4.69	2.08

Table 3 Comparison of two methods of deadlock detection ('con' and 'mono' are the numbers of constraints and monomials in the system of constraints, and 'vec/exp' the number of Unf_Σ-compatible vectors explored by the algorithm)

	PO					PO/ext			
	System size		Performance			System size		Performance	
Problem	cons	mons	vec/exp	$t[s]$	o-t-fly t $[s]$	cons	mons	vec/exp	$t[s]$
buf100	5051	14951	1	0.01	0.01	101	14951	1	0.02
mutual	497	2160	59	<0.01	<0.01	41	1152	149	0.01
ab_gesc	1200	9711	89	0.02	0.02	56	2837	330	0.04
sdl_arg	199	709	7	<0.01	<0.01	96	671	116	0.01
sdl_arg_d	223	834	9	<0.01	<0.01	92	762	60	<0.01
PEPRW(2)	147	917	13	<0.01	<0.01	36	445	22	<0.01
PEPRW(3)	1281	29062	73	0.01	0.01	63	42.10	107	0.02
PEPRW(4)	13513	2050563	469	0.33	0.76	100	46689	634	0.97
SEM(2)	32	92	7	<0.01	<0.01	15	76	8	<0.01
SEM(3)	86	344	16	<0.01	<0.01	22	228	17	<0.01
SEM(4)	216	1262	37	<0.01	<0.01	29	636	38	<0.01
SEM(5)	522	4760	86	<0.01	0.01	36	1696	87	0.01
SEM(6)	1228	18730	199	0.02	0.03	43	4372	200	0.05
SEM(7)	2830	76620	456	0.11	0.12	50	10968	457	0.23
SEM(8)	6416	322190	1033	0.49	0.54	57	26908	1034	1.38
SEM(9)	14354	1376528	2314	2.26	2.69	64	64800	2315	9.42
SEM(10)	31764	5923346	5131	10.74	13.65	71	153636	5132	56.10
PEPEL(1)	287	1064	11	<0.01	<0.01	50	1051	100	<0.01
PEPEL(2)	15366	228613	3986	1.85	3.06	109	50213	5773	10.42
STACK(3)	174	530	4	<0.01	<0.01	24	530	6	<0.01
STACK(4)	525	1610	5	<0.01	<0.01	30	1610	8	<0.01
STACK(5)	1578	4850	6	<0.01	<0.01	36	4850	10	<0.01
STACK(6)	4737	14570	7	0.01	0.01	42	14570	12	0.01
STACK(7)	14214	43730	8	0.05	0.06	48	43730	14	0.05
STACK(8)	42645	131210	9	0.14	0.20	54	131210	16	0.23
STACK(9)	127938	393650	10	0.41	0.61	60	393650	18	0.83
DME(2)	122	528	8	<0.01	<0.01	78	434	10	<0.01
DME(3)	321	1890	25	<0.01	<0.01	117	1119	33	<0.01
DME(4)	652	5536	56	0.01	0.01	156	2228	82	0.01
DME(5)	1145	14250	119	0.04	0.04	195	3845	192	0.05
DME(6)	1830	32832	246	0.10	0.11	234	6054	436	0.13
DME(7)	2737	68698	501	0.22	0.25	273	8939	970	0.35
DME(8)	3896	132480	1012	0.55	0.63	312	12584	2130	0.98
DME(9)	5337	238626	2035	1.25	1.48	351	17073	4632	2.59
DME(10)	7090	406000	4082	2.90	3.51	390	22490	10000	6.82
DME(11)	9185	658842	8177	6.68	8.12	429	28919	21462	17.42
RW(6)	397	2965	1	<0.01	<0.01	85	1255	13	<0.01
RW(9)	4627	141567	1	0.03	0.06	181	14059	19	0.01
RW(12)	49177	8501695	1	1.28	4.85	313	147877	25	0.26

are due to S. Melzer and S. Römer. The other examples are due to J.C. Corbett [5], after a translation into the PEP format by S. Melzer and S. Römer [19].

We used the PEP tool [2] to generate finite complete prefixes for our partial order algorithm, and for deadlock checking based on McMillan's method [16, 19] (the *McM* column in the tables), the *MIP* algorithm [19] (the *MIP* column) and the method based on computing stable models of a logic programs by K.Heljanko [10] (the *SM* column).

It is clear that the performance of the *MIP* algorithm highly depends on the performance of the tool used to solve the system of constraints. In fact, the algorithm in Section 7 can be considered as a specialised solver for (9), since the partial order and the conflict relation can be reconstructed from the constraints $M_{\text{in}} + \mathcal{C} \cdot x \geq 0$. To solve the system of constraints following the *MIP* approach, we used the `lp_solve` general purpose LP-solver by M.R.C.M. Berkelaar, since the CPLEX ^{TM} tool used in [19] is commercial. As CPLEX $^{\text{TM}}$ is considered to be more powerful than `lp_solve`, the results in the *MIP* column could be better.

The meaning of the columns corresponding to our *PO* algorithm is as follows: 'std' indicates the standard version of the algorithm using (11) as a system of constraints; 'ext' indicates the version using (22); and 'o-t-fly' indicates the on-the-fly version of our algorithm.

Table 3 contains the results of executing the algorithm in Section 7 using (11) and (22) as systems of constraints for deadlock detection (for the on-the-fly method only time is given, since it does not explicitly generate the system of constraints, and the number of explored Unf_Σ-compatible vectors is the same as for the original version of the algorithm). Note that the search space is usually greater for (22), because it does not allow as effective optimisation as (11), but since the size of the system (22) is often smaller, the actual running time of the algorithm is still acceptable. Moreover, memory savings for some of the examples are very significant. In view of the results in Table 3, the on-the-fly approach has a clear advantage, as it is not much slower than the original method, but uses much less memory and is easier to implement.

Although our testing was limited in scope, it appears that the algorithm proposed in this paper is fast, even for large prefixes. In [19], it has been pointed out that the *MIP* approach is good for 'wide' prefixes with a high number of cut-off events, whereas for prefixes with a small percentage of cut-off events McMillan's approach is better. It appears that our approach works well both for 'wide' prefixes with a high number of cut-off events and conflicts, and for 'narrow' ones with a high number of causal dependencies. The worst case is a prefix with a small number of conflicts and partial order dependencies (i.e., when nearly all pairs of events are in the $\|$ relation), combined with a small percentage of cut-off events. As the general problem is NP-complete in the size of the prefix, such examples can be artificially constructed (see, e.g., [17], where a reduction from the *3-SAT* problem is given), but we expect that the new algorithm should work well for practical verification problems.

Among the tested algorithms, the only comparable method was that based on a translation of a deadlock detection problem into a problem of finding a stable model of a logic program, proposed in [10]; the problem was then solved using the `smodels` tool [21, 23]. This tool is based on a backtracking search technique similar to Davis-Putnam method. An unusual feature of `smodels` is the ability to handle threshold-1 constraints (requiring that at most one variable from a given set is in the model), which were crucial for the efficiency of the method in [10]. After discussing this approach with its author, we concluded that if the logic solver used is powerful enough to model downclosing of configurations and freezing conflicting events in linear time (and this is the case for `smodels`) then the timing results of the *SM* method and our algorithm applied to the system (11) should be of the same order of magnitude. Indeed, the experimental results confirm that both methods are comparable, though they are based on different principles.

11. Conclusions

Experimental results indicate that the algorithm proposed in this paper can solve problems with more than a hundred thousand variables. As *MIP* problems with even a few hundreds of integer variables are usually a hard task for general purpose solvers, our approach overcomes

the existing limitations, while retaining the ability to compute all the minimal solutions and optimise w. r. t. a given monotonic cost function.

It is worth emphasising that earlier the limitation was not the size of computer memory, but rather the time to solve an NP-complete problem. With our method, the main limitation was the size of memory to store the system of constraints, but the on-the-fly approach overcomes this problem. The method presented in this paper, combined with the efficient parallel unfolding algorithm developed in [11, Section 5], yields an efficient unfolding-based framework for model checking Petri nets. Moreover, this framework can be extended to high-level Petri nets [11, Section 6].

Another contribution of this paper is a generalisation of *CDA*, which potentially can be used for various other applications. In particular, we show that many optimisation rules are compatible with *CDA*.

Acknowledgments We would like to thank Paul Watson for comments on an earlier version of this paper, Alexander Letichevsky and Sergei Krivoi for drawing our attention to *CDA* and discussions on the theory of Diophantine equations, Christian Stehno for help with using the PEP tool, and Stephan Melzer for sending his PhD thesis. In particular, we would like to thank Keijo Heljanko for explaining the *SM* algorithm, providing the mcsmodels deadlock checking tool and some of the unfoldings used in the experiments, and extensive discussions about net unfoldings.

This research was supported by the ORS Awards Scheme grant ORS/C20/4, the Epsrc grant GR/M99293, the Royal Academy of Engineering/Epsrc post-doctoral research fellowship EP/C53400X/1 (Davac) and the EC IST grant 511599 (Rodin).

References

1. Ajili F, Contejean E (1997) Avoiding slack variables in the solving of linear diophantine equations and inequations. Theor Comput Sci 173:183–208
2. Best E, Grahlmann B (1996) PEP —More than a Petri net tool. In: Proc TACAS'96, Lecture Notes in Computer Science, Springer-Verlag, Vol. 1055, pp 397–401
3. Clarke EM, Grumberg O, Peled D (1999) Model checking. MIT Press
4. Contejean E, Devie H (1994) An efficient incremental algorithm for solving systems of linear diophantine equations. Inf Comput 113:143–172
5. Corbett JC (1996) Evaluating deadlock detection methods for concurrent software. IEEE Trans Soft Eng SE-22(3):161–180
6. CPLEX Corporation (1995) CPLEX 3.0 Manual
7. Engelfriet J (1991) Branching processes of Petri nets. Acta Inform 28:575–591
8. Esparza J, Römer S, Vogler W (2002) An improvement of McMillan's unfolding algorithm. Form Method Syst Des 20(3):285–310
9. Esparza J (1994) Model checking based on branching processes. Sci Comput Program 23:151–195
10. Heljanko K (1999) Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets. IOS Press, Fundamenta Inform 37(3):247–268
11. Khomenko V (2003) Model checking based on prefixes of Petri net unfoldings. PhD Thesis, School of Computing Science, University of Newcastle
12. Khomenko V, Koutny M (2000) Deadlock checking using liner programming and partial order dependencies. Technical Report CS-TR-695, School of Computing Science, University of Newcastle
13. Khomenko V, Koutny M (2000) Verification of bounded Petri Nets using integer programming. Technical Report CS-TR-711, School of Computing Science, University of Newcastle
14. Koutny M, Best E (1999) Fundamental study: operational and denotational semantics for the box algebra. Theor Comput Sci 211:1–83
15. Krivoi S (1999) About some methods of solving and feasibility criteria of linear diophantine equations over the natural numbers domain (in Russian). Cybern Syst Anal 4:12–36
16. McMillan KL (1992) Using unfoldings to avoid state explosion problem in the verification of asynchronous circuits. In: Proc CAV'92, Lecture Notes in Computer Science, Springer-Verlag, Vol 663, pp 164–174
17. McMillan KL (1992) Symbolic model checking: an approach to the state explosion problem. PhD Thesis, School of Computer Science, Carnegie Mellon University

18. Melzer S (1998) Verifikation verteilter systeme mit linearer—und constraint-programmierung. PhD Thesis. Technische Universität München
19. Melzer S, Römer S (1997) Deadlock checking using net unfoldings. In: Proc CAV'97, Lecture Notes in Computer Science, Springer-Verlag, vol 1254, pp 352–363
20. Murata T (1989) Petri nets: properties, analysis and applications. In: Proc IEEE vol. 77 no. 4, pp. 541–580
21. Niemelä I, Simons P (1997) `smodels` —An implementation of the stable model and well-founded semantics for normal logic programs. In: Proc LPNMR'97, Lecture Notes in Artificial Intelligence, Springer-Verlag, vol 1265, pp 420–429
22. Silva M, Teruel E, Colom J-M (1998) Linear algebraic and linear programming techniques for the analysis of place/transition net systems. In: Reisig W, Rozenberg G (eds) Lectures on Petri nets I: basic models. Springer-Verlag, pp 309–373
23. Simons P (1999) Extending the stable model semantics with more expressive rules. In: Proc LPNMR'99, Lecture Notes in artificial intelligence, Springer-Verlag vol 1730.