Linear Programming Deadlock Checking Using Partial Order Dependencies

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

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 net [14]. 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. We adopt Cornerstone'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. 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. Experimental results demonstrate that the new technique achieves significant speedups.

Keywords: Model checking, integer linear programming, Petri nets, unfolding, causality and concurrency.

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 behaviours comes from highly complicated interactions with the external environment rather than from complicated data structures and manipulations thereof. One way of coping with this complexity problem is to use formal methods and, especially, computer-aided verification tools implementing model checking [4,8] — a technique in which the verification of a property 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 help in coping with this, a number of techniques have been proposed which can roughly be classified as aiming at an implicit compact representation of the full state space of a reactive concurrent system, or at an explicit representation of a reduced (though sufficient for a given verification task) state space of the system. Techniques aimed at reduced representation of state spaces are typically based on the independence (commutativity) of some actions, often relying on the partial order view of concurrent computation. Such a view is the basis for algorithms employing McMillan's unfoldings [11,12,14], where the entire state space of a Petri Net is represented implicitly using an acyclic net to represent a system's actions and local states. The unfolding technique presented in [14,15] reduces memory requirements, but the deadlock checking algorithm proposed are quite slow, even for medium-size unfoldings.
In [15], the problem of deadlock checking a Petri net was reduced to a linear integer programming problem. In this paper, we present a further development of this approach. We adopt the Contejean-Davie's algorithm ([1, 2, 3-7]) for efficiently solving systems of linear constraints over the domain of natural numbers. We refine this algorithm by employing unfolding-specific properties of the systems of linear constraints to be solved, in model checking aimed at deadlock detection. The essence of the proposed modifications is to transfer the information about causality and conflict between events involved in an unfolding into a relationship between the corresponding integer variables in the system of linear constraints. The results of initial experiments demonstrate that the new technique achieves 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 [15] 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, 2, 3-7] and recalls the main aspects of the Contejean-Davie's Algorithm (CDA) for solving systems of linear constraints over the natural number domain. The algorithm we propose in this paper is a variation of CDA, developed specifically to exploit partial order dependencies between events in the unfolding of a Petri net. Our algorithm is described in Section 5; we provide theoretical background, model heuristics, and outline ways of reducing the number of variables and constraints in the original system presented in [15]. Section 6 contains results of experiments obtained for a number of benchmark examples. Section 7 briefly discusses the parallelisation aspects of the new algorithm, and in Section 9, we discuss how the approach can be generalised to deal with other relevant verification problems, such as mutual exclusion, controllability and reachability analysis. Section 8 describes possible directions for future research.

2 Basic definitions

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

Petri net. A net is a triple \( N = (S, T, F) \) such that \( S \) and \( T \) are disjoint sets of respectively places and transitions, and \( F \subseteq (S \times T) \cup (T \times S) \) is a flow relation. A marking of \( N \) is a multiset \( M \) of places, i.e. \( M : S \rightarrow \mathbb{N} \). 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 \( s = \{ (s, x) \in F \} \) and \( t = \{ (x, y) \in F \} \), for all \( x \in S \cup T \). We will assume that \( s \neq \emptyset \neq t \), for every \( s \in T \).

A net system is a pair \( \Sigma = (N, M_0) \) 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 \( M[t] \), if for every \( a \in t, M(a) \geq 1 \). Such a transition can be executed, leading to a marking \( M' \) defined by \( M' = M - \{a \mid a \in t\} + \{a \mid a \in t\} \). We denote this by \( M[t] M' \) or \( \text{M}(t) \text{M}' \). The set of reachable markings of \( \Sigma \) is the smallest set \( \{M_0\} \) containing \( M_0 \) such that if \( M \in \{M_0\} \) and \( M[t] M' \), then \( M' \in \{M_0\} \). For a finite sequence of transitions, \( \sigma = t_1 \ldots t_n \), we denote \( \text{M}(t_1 \ldots t_n) \text{M} \) if there are markings \( M_1, \ldots, M_n \) such that \( M_n = M_0 \) and \( \text{M}(t_1 \ldots t_{n-1}) \text{M}_n \), for \( \delta = 1, \ldots, n \).

A marking is deadlocked if it does not enable any transitions. The net system \( \Sigma \) is deadlock-free if no reachable marking is deadlocked; safe if for every reachable marking \( M', M'(S) \subseteq \{0, 1\} \); and bounded if there is \( k \in \mathbb{N} \) such that \( M(S) \subseteq \{0, \ldots, k\} \), for every reachable marking \( M \).
Marking equation. Let $\Sigma = (N, M_0)$ be a net system, and $S = \{s_1, \ldots, s_n\}$ and $T = \{t_1, \ldots, t_m\}$ be sets of input places and transitions, respectively. We will often identify a marking $M$ of $\Sigma$ with a vector $M = (M(s_1), \ldots, M(s_n))$ such that $M(s_i) = M_i$ for all $i \leq n$. The incidence matrix of $\Sigma$ is an $m \times n$ matrix $K = (K_{ij})$ such that, for all $i \leq m$ and $j \leq n$,

$$K_{ij} = \begin{cases} 1 & \text{if } a \in \delta_i \setminus \delta_j \\ -1 & \text{if } a \in \delta_j \setminus \delta_i \\ 0 & \text{otherwise} \end{cases}$$

The Pnueli vector of a finite sequence of transitions $s$ is a vector $s_m = (s_1, \ldots, s_m)$ such that $s_i$ is the number of the occurrences of $s_i$ within $s$, for every $i \leq n$. One can show that if $s$ is an execution sequence such that $M_0(s), M_0$ then $M = M_0 + M_s$. This provides a motivation for investigating the feasibility (or solvability) of the following system of equations:

$$\begin{align*}
\mathbf{M} &= M_0 + M_s \\
\mathbf{M} &\in M^n \text{ and } s \in M^n
\end{align*}$$

If we fix the marking $M$, then the feasibility of the above system is a necessary condition for $M$ to be reachable from $M_0$.

Branching processes. Two nodes of a net $N = (S, T, F)$, $y$ and $y'$, are in conflict, denoted by $y \not\sim y'$, if there are distinct transitions $t, t' \in T$ such that $t \cap t' = \emptyset$ and $(t, y)$ and $(t', y')$ are in the reflexive transitive closure of the flow relation $F$; denoted by $\leq$. A node $y$ is self-conflict if $y \not\sim y$.

An occurrence net is a net $ON = (B, F, G)$ where $B$ is the set of conditions (places) and $F$ is the set of events (transitions). It is assumed that $ON$ is acyclic (i.e., $\leq$ is a partial order); for every $b \in B$, $\lvert b \rvert \leq 1$; for every $y \in B \cup F$, $\lnot(y \not\sim y)$ and there are finitely many $y$ such that $y \not\sim y$, where $\not\sim$ denotes the reflexive transitive closure of $\leq$. $\text{Min}(ON)$ will denote the minimal elements of $B \cup F$ with respect to $\leq$. The relation $\not\sim$ is the causality relation. Two nodes are co-related, denoted by $y \sim y'$, if neither $y \not\sim y'$ nor $y' \not\sim y$.

A homomorphism from an occurrence net $ON$ to a net system $\Sigma$ is a mapping $h : B \cup F \to \Sigma \cup T$ such that $h(B) \subseteq S$ and $h(F) \subseteq T$, for all $a \in F$, the restriction of $h$ to $a$ is a bijection between $a$ and $h(a)$; the restriction of $h$ to $a$ is a bijection between $a$ and $h(a)$; the restriction of $h$ to $a$ is a bijection between $a$ and $h(a)$; the restriction of $h$ to $a$ is a bijection between $a$ and $h(a)$; and for all $a, f, g \in F$, if $a \not\sim a'$ and $h(a) = h(a')$ then $a = a'$.

A branching process of $\Sigma$ [10] is a quadruple $\pi = (B, F, G, h)$ such that $(B, F, G)$ is an occurrence net and $h$ is a homomorphism from $ON$ to $\Sigma$. A branching process $\pi' = (B', F', G', h')$ of $\Sigma$ is a profile of a branching process $\pi = (B, F, G, h)$, denoted by $\pi' \subseteq \pi$, if $(B', F', G')$ is a subset of $(B, F, G)$ such that if $e \in F$ and $(e, a) \in G$ then $a \in B'$; if $b \in B'$ and $(a, b) \in G$ then $a \in F'$; and $h'$ is the restriction of $h$ to $B' \cup F'$. For each $\Sigma$ there exists a unique (up to isomorphism) maximal (w.r.t. $\sqsubseteq$) branching process, called the unfolding of $\Sigma$.

Configurations and cuts. A configuration of an occurrence net $ON$ is a set of events $C$ such that for all $a, f \in C$, $-a \not\sim f$ and, for every $a \in C$, $f \not\sim a$ implies $f \in C$. A cut is a maximal w.r.t. set inclusion set of conditions $C$ such that $h(C, b)$ for all $b, h \in B$. Every marking reachable from $\text{Min}(ON)$ is a cut.

Let $C$ be a finite configuration of a branching process $\pi$. Then $\text{Out}(C) = (\text{Min}(ON) \cup C') \setminus C$ is a cut; moreover, the multiset of places $h(\text{Out}(C))$ is a reachable marking of $\Sigma$, denoted $\text{Mark}(C)$. A marking $M$ of $\Sigma$ is represented in $\pi$ if the latter contains
a finite configuration $C$ such that $M = \text{Mark}(C)$. Every marking represented in $\hat{\nu}$ is reachable, and every reachable marking is represented in the unfolding of $\hat{\nu}$.

A branching process $\hat{\nu}$ is complete if for every reachable marking $M'$ of $\hat{\nu}$, there is a configuration $C$ in $\nu$ such that $\text{Mark}(C) = M'$, and for every transition $t$ enabled by $M'$, there is a configuration $C_t(e)$ such that $e \in C$ and $a(e) = \nu_t$.

Although, in general, an unfolding is infinite, for every bounded net system $\nu$ one can construct a finite complete prefix $\mathcal{U}_{\nu_{\text{BR}}}$ of the unfolding of $\nu$. Moreover, there are so-called cut-off events in $\mathcal{U}_{\nu_{\text{BR}}}$ such that, for every reachable marking $M'$ of $\hat{\nu}$, there exists a configuration $C$ in $\mathcal{U}_{\nu_{\text{BR}}}$ such that $M = \text{Mark}(C)$ and no event in $C$ is a cut-off event.

3 Deadlock detection using linear programming

In the rest of this paper, we will assume that $\mathcal{U}_{\nu_{\text{BR}}} = (F, E, \mathcal{C}, A)$ is a finite complete prefix of the unfolding of a bounded net system $\nu = (S, F, E, A)$. We will denote by $M_{\text{in}}$ the canonical initial marking of $\mathcal{U}_{\nu_{\text{BR}}}$ which places a single token in each of the minimal conditions and no token elsewhere. Furthermore, we will assume that $b_1, b_2, \ldots, b_p$ and $a_1, a_2, \ldots, a_q$ are respectively the conditions and events of $\mathcal{U}_{\nu_{\text{BR}}}$, and that $C$ is the $p \times q$ incidence matrix of $\mathcal{U}_{\nu_{\text{BR}}}$. The set of cut-off events of $\mathcal{U}_{\nu_{\text{BR}}}$ will be denoted by $\mathcal{E}_{\text{cut}}$.

We now recall the main results from [18]. A finite and complete prefix $\mathcal{U}_{\nu_{\text{BR}}}$ may be treated as an acyclic safe net system with the initial marking $M_{\text{in}}$. Each reachable deadlocked marking in $\nu$ is represented by a deadlocked marking in $\mathcal{U}_{\nu_{\text{BR}}}$. However, some deadlocked markings of $\mathcal{U}_{\nu_{\text{BR}}}$ may be beyond the cut-off events and may not correspond to deadlocks in $\nu$. Such deadlocks can be excluded 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.

**Theorem 1.** ([18]) $\nu$ is deadlock-free if and only if the following system has no solution (in $M'$ and $\nu$):

\[
\begin{cases}
M' = M_{\text{in}} + C \cdot \nu \\
\sum_{h \in E} M'(h) \leq |E| - 1 & \text{for all } e \in F \\
\sum_{h \in E} a(h) = 0 & \text{for all } e \in \mathcal{E}_{\text{cut}} \\
M' \in \mathbb{N}_r & \text{and } \nu \in \mathbb{N}_r
\end{cases}
\]

where $a(e) = \nu$, for every $i \leq q$.

In order to decrease the number of integer variables, $M' \geq 0$ can be treated as a rational vector since $\nu \in \mathbb{N}_r$ and $M' = M_{\text{in}} + C \cdot \nu \geq 0$ always imply that $M' \in \mathbb{N}_r$. Moreover, as an event can occur at most once in a given execution sequence of $\mathcal{U}_{\nu_{\text{BR}}}$, from the initial marking $M_{\text{in}}$, it is possible to require that $\nu$ be a binary vector, $\nu \in \{0, 1\}^r$.

To solve the mixed-integer LP-problem (1), [18] used the general-purpose LP-solver CPLEX [8], and demonstrated that there are significant performance gains if the number of cut-off events is relatively high since all variables in $\nu$ corresponding to cut-off events are set to 0.

---

1 Initially, cut-off events are nodes at which the potentially infinite unfolding may be cut without losing any essential information about the behaviour of $\nu$; see [10–12, 14, 15] for details.
Remarks: The characterisation of the deadlock checking problem given by (1) suggests that, in general, one can consider the following specific problems:

- Pure feasibility test (to discover whether a net system is deadlock-free).
- Checking feasibility and, in the case that there are deadlocks, to find an execution sequence leading to a deadlock.
- To facilitate debugging, finding a shortest path leading to a deadlock. This results in an optimisation problem, i.e. minimise $L(x) = x(s_1) + \cdots + x(s_p)$ under the constraints given by (1).

We will show in section 5.1 that it is possible to reduce (1) to a pure integer LP-problem without increasing the total number of integer variables. Moreover, (1) 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 seems 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, 2, 3-7], 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) [3-7] solves a system of linear homogeneous equations with arbitrary integer coefficients:

$$
\begin{align*}
0 &= a_1 x_1 + \cdots + a_{p1} x_p \\
0 &= a_2 x_1 + \cdots + a_{p2} x_p \\
&\vdots \\
0 &= a_{p} x_1 + \cdots + a_{pp} x_p \\
\end{align*}
$$

or $A \cdot x = 0$ where $x \in \mathbb{Z}^p$ and $A = (a_{ij})$. For every $1 \leq j \leq q$, let

$$
\epsilon_j = (0, \ldots, 0, 1, 0, \ldots, 0)
$$

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

$$
a(x) = \begin{pmatrix}
a_1 x_1 + \cdots + a_{p1} x_p \\
a_2 x_1 + \cdots + a_{p2} x_p \\
\vdots \\vdots \\
a_p x_1 + \cdots + a_{pp} x_p \\
\end{pmatrix} = x_1 \cdot a(s_1) + \cdots + x_p \cdot a(s_p),
$$

where $a(s_j)$ — the $j$-th column vector of the matrix $A$ — is called the $j$-th basic default vector.

The set $S$ of all solutions of (2) can be represented by a finite basis $B$ which is the minimal (w.r.t. set inclusion) subset of $S$ such that every solution is a linear combination with non-negative integer coefficients of the solutions in $B$. It can be shown that $B$ comprises all solutions in $S$ different from the trivial one, $x = 0$, which are minimal with respect to the $\leq$ ordering on $\mathbb{Z}^q$ (if $x \leq x'$, $x \leq x'$, for all $x \leq y$, moreover, $x \leq x'$ if $x \leq x'$ and $x \neq x'$).
The representation (3) suggests that any solution of (2) can be seen as a multiset of
default vectors whose sum is 0. Choosing these vectors in an arbitrary order amounts
to constructing a sequence of default vectors starting from, and returning to, the origin
of \mathbb{R}^r. 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.

A vector \( \mathbf{e} \in \mathbb{R}^r \) (corresponding to a sequence of default vectors) such that
\( a(\mathbf{e}) \neq 0 \) can be incremented by 1 on its \( j \)-th component provided that
\( a(n \cdot \mathbf{e}) - a(x) = a(x) + a(\mathbf{e}_j) \) lies in the half-space containing 0 and delimited by
the affine hyperplane perpendicular to the vector \( a(\mathbf{e}) \) at its extremity when originating
from 0 (see Figure 1).

\[
\begin{align*}
\text{Fig. 1. Geometric interpretation of the branching condition in CDA.}
\end{align*}
\]

This reflects a view that \( a(\mathbf{e}) \) should not become too large, hence adding \( a(\mathbf{e}_j) \) to \( a(\mathbf{e}) \)
should yield a vector \( a(n \cdot \mathbf{e}) = a(x) + a(\mathbf{e}_j) \) returning to the origin. Formally, this
restriction can be expressed by saying that given \( \mathbf{e} = (e_1, \ldots, e_r) \),

\[
\text{increment by 1 an } e_j \text{ satisfying } a(e_j) a(\mathbf{e}_j) < 0 ,
\]

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 (4).

**Theorem 2.** ([77]) The following hold for the CDA shown in Figure 1:

1. Every minimal solution of the system (3) is computed. \( \text{(completeness)} \)
2. Every solution computed by CDA is minimal. \( \text{(soundness)} \)
3. The algorithm always terminates. \( \text{(termination)} \)

Figure 3 illustrates the process of solving the homogeneous system of linear equations
considered in [13]:

\[
\begin{align*}
\begin{cases}
-x_1 + x_2 + 2x_3 - 3x_4 = 0 \\
x_1 + 3x_3 - 2x_4 = 0
\end{cases}
\end{align*}
\]
- Search breadth-first a directed cyclic graph rooted at $e_1, \ldots, e_r$
- If a node $y$ is equal to, or greater than, an already encountered solution of $A \cdot x = 0$ then $y$ is a terminal node
- Otherwise construct the sum of $y$ by computing $y + e_j$ for each $j \leq q$ satisfying $a(y) \cap a(e_j) \neq \emptyset$

**Fig. 2. CBA (breadth-first version)**

![Diagram](image)

**Fig. 3. Search graph constructed by CBA in Figure 2: 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 $x' = (0, 1, 1)$ and $x'' = (4, 2, 1, 0)$ are two minimal solutions**
The example shows redundancies; some vectors were computed more than once. This can be remedied by using frozen components, defined thus: assume that there is a total ordering $\alpha_\circ$ on the set of each node $x$ of the search graph constructed by CDA.

If $x + e$ and $x + e'$ are two distinct sums of a node $x$ such that $x + e; \alpha_\circ x + e'$, then the $i$-th component is frozen in the sub-graph rooted at $x + e'$ and cannot be incremented even if (4) is satisfied.

The modified algorithm is still complete (7), and builds a forest which is a sub-graph of the original search graph.

By defining the ordering $\alpha_\circ$ as $x + e; \alpha_\circ x + e'$ if $i < j$ we obtain, for the system in the above example, the graph shown in figure 4 (see [18]).

---

Fig. 4. Search graph constructed by the ordered version of CDA; frozen components are underlined, and the + indicate nodes which cannot be developed according to condition (4) and frozen components are.

---

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

$-x' \leq x$. Then, instead of starting with the vectors $e_1, \ldots, e_p$, the algorithm starts with $x'$. The rest of the operation remains the same, but the minimal elements of

---

*The ordering $\alpha_\circ$ may be defined in other ways as well (see [7]).
the set \( \mathcal{S} = \{ \mathbf{z} \mid A \cdot \mathbf{z} = 0 \land \mathbf{z} ' \leq \mathbf{z} \} \) do not give all the solutions of
\[
\begin{align*}
A \cdot \mathbf{z} &= 0 \\
\mathbf{z}' &\leq \mathbf{z}
\end{align*}
\]

However, any solution of the above system can be represented as a sum of a minimal element of \( \mathcal{S} \) and a non-negative integer linear combination of minimal solutions of the original system.

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

With the above extensions, CPA allows one to solve non-homogeneous diophantine systems
\[
\begin{align*}
\begin{cases}
d_1 \mathbf{z}_1 + \ldots + d_p \mathbf{z}_p &= \mathbf{d}_1 \\
d_2 \mathbf{z}_1 + \ldots + d_p \mathbf{z}_p &= \mathbf{d}_2 \\
\vdots & \vdots \\
d_q \mathbf{z}_1 + \ldots + d_p \mathbf{z}_p &= \mathbf{d}_q
\end{cases}
\end{align*}
\]

By adding a new variable, \( \mathbf{z}_0 \), we can transform (5) into a homogeneous system
\[
\begin{align*}
\begin{cases}
-d_1 \mathbf{z}_0 + d_1 \mathbf{z}_1 + \ldots + d_p \mathbf{z}_p &= 0 \\
-d_2 \mathbf{z}_0 + d_2 \mathbf{z}_1 + \ldots + d_p \mathbf{z}_p &= 0 \\
\vdots & \vdots \\
-d_q \mathbf{z}_0 + d_q \mathbf{z}_1 + \ldots + d_p \mathbf{z}_p &= 0
\end{cases}
\end{align*}
\]

Let \( \mathcal{M}_k (k = 0, 1) \) be the set of all minimal solutions \( \mathbf{z} = (\mathbf{z}_0, \mathbf{z}_1, \ldots, \mathbf{z}_p) \) of this system with \( \mathbf{z}_0 = k \). Then any solution of (5) can be represented as
\[
\mathbf{z} = \mathbf{y} + \sum_{\mathbf{z} \in \mathcal{M}_k} \mathbf{c}_z \mathbf{z} ',
\]
where \( \mathbf{y} \in \mathbb{N} \) and each \( \mathbf{c}_z \) belongs to \( \mathbb{N} \). Thus to solve (5) it suffices to add just one variable which is frozen as soon as it reaches the value 1.

The task of solving a system of linear inequalities
\[
\begin{align*}
\begin{cases}
d_1 \mathbf{z}_1 + \ldots + d_p \mathbf{z}_p &\leq \mathbf{d}_1 \\
d_2 \mathbf{z}_1 + \ldots + d_p \mathbf{z}_p &\leq \mathbf{d}_2 \\
\vdots & \vdots \\
d_q \mathbf{z}_1 + \ldots + d_p \mathbf{z}_p &\leq \mathbf{d}_q
\end{cases}
\end{align*}
\]

is more complicated. The standard linear programming approach is to reduce it to a
system of equations
\[
\begin{align*}
\begin{cases}
d_1 \mathbf{z}_1 + \ldots + d_p \mathbf{z}_p + y_1 &= \mathbf{d}_1 \\
d_2 \mathbf{z}_1 + \ldots + d_p \mathbf{z}_p + y_2 &= \mathbf{d}_2 \\
\vdots & \vdots \\
d_q \mathbf{z}_1 + \ldots + d_p \mathbf{z}_p + y_p &= \mathbf{d}_q
\end{cases}
\end{align*}
\]

Some solutions of (5) cannot be represented as non-negative linear combinations of minimal solutions, and Ait and Cassin are the notion of a non-decomposable solution to deal with this problem [2]. For our purposes, however, it is sufficient to check only the minimal solutions.
by adding slack variables \( y \in \mathbb{N} \), but this transformation increases the number of variables from \( n \) to \( n + p \). And, as the computation time can grow exponentially in the number of variables, such an approach is not efficient. Moreover, the slack variables may assume arbitrary values in \( \mathbb{N} \), even if all the variables in the original problem are binary as in (1); as a result, the search space grows very rapidly.

A better approach is to deal with inequalities directly. It has been developed in [1, 2], where CDA was generalized to solve non-homogeneous systems of equations and inequalities. (A system of non-homogeneous inequalities can be reduced to a homogeneous one by adding just one binary variable, similarly as described above for (3); it is also possible to slightly change the algorithm to solve non-homogeneous systems directly.) For a system of linear constraints \( A \cdot x = 0 \land A' \cdot x \leq 0 \), the branching condition (4) is modified by saying that given \( x = (x_1, \ldots, x_p) \)

increment by 1 an \( x_i \) for which there exist \( y_1, \ldots, y_n \) such that the vector \( (x_1, \ldots, x_p, y_1, \ldots, y_n) \) can be incremented on its \( i \)-th component according to (4) applied to the system \( A' \cdot x + y = 0 \), where \( i \) is the number of rows in \( A' \) and \( y = (y_1, \ldots, y_n) \).

As shown in [1, 2], this condition can be expressed as:

\[
(A \cdot x) \otimes (A' \cdot x) + \sum_{j=1}^{n} \min_j((A'_j \otimes x)(A'_j \otimes c_j), \max_j(0, (A'_j \otimes x)(A'_j \otimes c_j))) < 0
\]

where \( A'_j \) is the \( j \)-th row of \( A' \). To ensure termination in the general case, [1, 2] add one more condition, but if the variables are bounded this is always guaranteed.

5 An algorithm for deadlock detection

In this section, we introduce our deadlock checking algorithm, explain its theoretical background, and present some useful heuristics.

5.1 Reducing to a pure integer problem

The problem (1) can be reduced to a pure integer one, by substituting the expression for \( M' \) given by the matching equation into other constraints and, at the same time, reducing the total number of constraints. Each equation in \( M' = M_{\text{con}} + C' \cdot x \) has the form

\[
M'(b) = M_{\text{con}}(b) + \sum_{f \in e} \pi(f) - \sum_{f \in e} \pi(f) \quad \text{for } b \in \mathbb{N}.
\]  

(7)

After substituting these into (1) we obtain the system

\[
\begin{cases}
\sum_{b \in e} \left( \sum_{f \in e} \pi(f) - \sum_{f \in e} \pi(f) \right) \leq |e| - 1 - \sum_{b \in e} M_{\text{con}}(b) \quad \text{for all } e \in E \\
M_{\text{con}}(b) + \sum_{f \in e} \pi(f) - \sum_{f \in e} \pi(f) \geq 0 \quad \text{for all } b \in \mathbb{N} \\
x \in \{0, 1\}^r \text{ and } x(d) = 0 \text{ for all } d \in \mathbb{N}
\end{cases}
\]  

(8)

Usually, each inequality in (8) contains relatively few variables, and the size of memory required to store the entire system is linear (rather than quadratic) in the size of \( \mathcal{U} \mathcal{D} \).

As (8) is a pure integer problem, CDA is directly applicable. However, since the number of variables can be large, it needs further refinement.
5.2 Partial-order dependencies between variables

In [15], a finite prefix of the unfolding is used only for building a system of constraints, and the latter is then passed to the 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 be easily derived from \( \mathcal{G}_{\mathcal{M}} \). For example, if we set \( \varepsilon(a) = 1 \) then each \( \varepsilon(f) \) such that \( f \) is a predecessor (in causal order) of \( a \) must be equal to 1, and each \( \varepsilon(g) \) such that \( g \) is in conflict with \( a \) must be equal to 0. Similarly, if we set \( \varepsilon(a) = 0 \) then no event \( f \) for which \( a \) is a cause can be executed in the same run, and so \( \varepsilon(f) \) should be equal to 0. Our algorithm will use these observations to reduce the search space, and the experimental results indicate that taking into account causal dependencies, in combination with some useful heuristics, can lead to significant speedups.

**Definition 1.** A vector \( \varepsilon \in \{0, 1\}^n \) is compatible with \( \mathcal{G}_{\mathcal{M}} \) if for all distinct events \( a, f \in \mathcal{E} \) such that \( \varepsilon(a) = 1 \), we have:

\[
f \prec a \Rightarrow \varepsilon(f) = 1 \quad \text{and} \quad f \not\prec a \Rightarrow \varepsilon(f) = 0 .
\]

The motivation for considering compatible vectors follows from the next result.

**Theorem 3.** A vector \( \varepsilon \in \{0, 1\}^n \) is compatible with \( \mathcal{G}_{\mathcal{M}} \) if and only if there exists an execution sequence starting at \( M_{\mathcal{M}} \), whose Parish vector is \( \varepsilon \).

**Proof.** (\( \Rightarrow \)) Let \( \sigma \) be an execution sequence starting from \( M_{\mathcal{M}} \). For \( a \in \mathcal{E} \) to be executed, it is necessary for all \( f \in \mathcal{E} \) satisfying \( f \prec a \) to occur before \( a \). Moreover, if \( f \not\prec a \) then \( f \) cannot happen in the same execution sequence. Hence the Parish vector of \( \sigma \) is compatible with \( \mathcal{G}_{\mathcal{M}} \).

(\( \Leftarrow \)) For each compatible vector \( \varepsilon \), it is possible to build an execution sequence whose Parish vector is \( \varepsilon \). Indeed, as shown in [15], for acyclic nets the feasibility of marking equation is a sufficient condition for a marking to be reachable. Moreover, the proof of this result presented in [15] implies that any solution of this equation corresponds to at least one execution sequence \( \sigma \). Hence, if for a given compatible vector \( \varepsilon \),

\[
M = M_{\mathcal{M}} + C \cdot \varepsilon \geq 0
\]

then \( M \) is reachable and there is an execution sequence leading to \( M \) whose Parish vector is \( \varepsilon \). Therefore it is enough to show that for every compatible vector \( \varepsilon \), \( M_{\mathcal{M}} + C \cdot \varepsilon \geq 0 \), i.e.

\[
M_{\mathcal{M}}(b) + \sum_{f \in \mathcal{E} \setminus b} \varepsilon(f) - \sum_{f \in \mathcal{E} \setminus b} \varepsilon(f') \geq 0 \quad \text{for all } b \in \mathcal{B} .
\]

Since \( |\mathcal{B}| \leq 1 \), for all \( b \in \mathcal{B} \),

\[
\sum_{f \in \mathcal{E} \setminus b} \varepsilon(f) = \begin{cases} 0 & \text{if } b = \emptyset \\ \varepsilon(f') & \text{if } b = \{f'\} \end{cases}
\]

and there are two possible cases:

**Case 1:** \( b = \emptyset \). Then \( b \) is an initial condition and so \( M_{\mathcal{M}}(b) = 1 \). In this case \( (3) \) has the form \( \sum_{f \in \mathcal{E} \setminus b} \varepsilon(f) \leq 1 \) and it holds true due to the fact that all the events in \( M \) are in conflict.

**Case 2:** \( b = \{f'\} \) for some \( f' \in \mathcal{E} \). Then \( M_{\mathcal{M}}(b) = 0 \), so \( (3) \) has the form \( \sum_{f \in \mathcal{E} \setminus b} \varepsilon(f) \leq \varepsilon(f') \) and it also holds true since all the events in \( M \) are in conflict, and \( f' \) is a predecessor for all of them. \( \square \)
Corollary 1. For each reachable marking \( M' \) of \( S \), there exists an execution sequence in \( \text{Unf}_M \) leading to a marking representing \( M' \), whose Parikh vector \( \pi \) is compatible with \( \text{Unf}_M \) and \( \pi(a) = 0 \), for every \( a \in \text{Exit} \).

Proof. Each reachable marking \( M' \) of \( S \) is represented in \( \text{Unf}_M \) by a marking \( M'' \) which can be reached from \( M' \) through an execution sequence \( \sigma \) without cut-off events. Theorem 3 implies that the Parikh vector of \( \sigma \) is compatible with \( \text{Unf}_M \). \( \square \)

In view of the last result, it is sufficient for a deadlock detection algorithm to check only compatible vectors, whose components corresponding to cut-off events are equal to zero. This can be done by building minimal compatible closure of a vector \( \pi \) (see the definition below) in each step of CDA and freeing all \( \pi(a) \) such that \( a \in \text{Exit} \).

Definition 2. A vector \( \pi \in \{0,1\}^n \) has a compatible closure if and only if for all \( a, f \in \text{Exit} \), \( \pi(f) = 1 \) implies \( \pi(a) = 0 \). If \( \pi \) has a compatible closure then its minimal compatible closure exists and is unique. Moreover, in such a case if \( \pi \) has zero components for all cut-off events, then the same is true for its minimal compatible closure.

Let us consider the causal ordering \( <_c \), \( \preceq_c \), \( \succeq_c \), and \( <_{cc} \), \( \succeq_{cc} \) (see figure 5), and \( \pi = (1,0,1,0) \). Then \( \pi' = (1,1,1,0) \) and \( \pi'' = (1,1,1,1) \) are compatible closures of \( \pi \), and \( \pi' \) is the minimal one.

\[ \text{Fig. 5. An occurrence net} \]

Theorem 4. A vector \( \pi \in \{0,1\}^n \) has a compatible closure if and only if for all \( a, f \in \text{Exit} \), \( \pi(f) = 1 \) implies \( \pi(a) = 0 \). If \( \pi \) has a compatible closure then its minimal compatible closure exists and is unique. Moreover, in such a case if \( \pi \) has zero components for all cut-off events, then the same is true for its minimal compatible closure.

Proof. Straightforward. We just point out that to build the minimal compatible closure of \( \pi \) it is enough to set to 1 all the components \( \pi(f) \) for which there is a such that \( f \prec_c a \) and \( \pi(a) = 1 \). \( \square \)

It may happen that a vector \( \pi \) has a compatible closure according to theorem 4, but it cannot be computed because some of the zero components of \( \pi \) to be set to 1 have been frozen. In this case, the algorithm should behave as if such a closure cannot be built.

Branching condition. Each step of CDA can be seen as moving from a point \( d(f) \) along a default vector \( d(f) \) such that \( d(f) \cap d(f) < 0 \), which is interpreted as returning to the origin (see figure 1). However, for an algorithm checking compatible vectors only, each step is moving along a vector which may be represented as a sum of several default vectors, and such a geometric interpretation is no longer valid. Indeed, let us consider the same ordering as in figure 5, and the equation

\[ d(f) = x_1 + 3x_2 - 3x_3 - 3x_4 = 0 \]
which has a solution \( x = (1, 1, 1, 1) \) with an initial constraint \( x_1 = 1 \). Then we start the algorithm from the vector \( x = (1, 0, 0, 0) \), and the sequence of steps should begin from either \( c_5 \) or \( c_5 + c_4 \) or \( c_5 + c_4 \). But \( a(c_5) \otimes a(c_5) = -\lambda < 0 \), \( a(c_5) \otimes a(c_5 + c_4) = 2 \beta < 0 \), and \( a(c_5) \otimes a(c_5 + c_4) = 2 \beta < 0 \), so we cannot choose a vector to make the first step! A possible solution is to interpret each step \( c_5 \) as a sequence of smaller steps \( c_5, \ldots, c_n \) where we choose only the first element \( c_5 \), for which \( a(c_5) \) does return to the origin, and then add the remaining one in order for \( x + c_5, \ldots, c_n \) to be compatible, without worrying where they actually land, as shown in figure 6. (If there is a compatible closure of \( x + c_5 \) then \( c_5 \) cannot be chosen.) This means that we check the condition \( a(c_5) \otimes a(c_5) < 0 \) which coincides with the original CDA’s branching condition. Though we are moving along positively different vector. As the following theorem shows, this technique is still complete.

![Fig. 6: Geometric interpretation of the new branching condition](image)

**Theorem 2.** Every non-trivial minimal compatible solution can be computed using the above method.

**Proof.** The proof is adopted from [7]. We assume that the constraints are expressed as a system of linear homogeneous equations \( A \cdot x = 0 \), i.e. (2). The argument can then be generalised to an arbitrary system of linear constraints using the approach described in [1, 2].

Let \( x \) be a minimal non-trivial solution of \( A \cdot x = 0 \) compatible with \( \text{Unf}_p \). The algorithm needs to construct a sequence of vectors \( x_1, \ldots, x_n \) such that \( x < x_{n+1}, x = x_1 \) and each \( x_i \) is compatible with \( \text{Unf}_p \). For \( x_1 \), the minimal compatible closure of any \( x \) such that \( c > x \) can be chosen (such an \( x \) does exist and has a compatible closure since \( x \) is non-trivial and can be seen as — possibly not minimal — compatible closure of some \( c \)). Suppose that the algorithm has already constructed \( x_1, \ldots, x_n \) and \( x_n \leq x \). Then there exists \( x_{n+1} \) such that \( x = x_n + x_{n+1} \) and

\[
0 = \|a(x)\|^2 = \|a(x_n)\|^2 + \|a(x_{n+1})\|^2 + 2a(x_n) \otimes a(x_{n+1}) ,
\]

where \( \| \cdot \| \) denotes the Euclidean norm in \( \mathbb{R}^p \). Hence \( a(x_n) \otimes a(x_{n+1}) < 0 \). Consequently, there exists \( \delta \leq \alpha \) such that \( x_{n+1} \leq x \) and \( a(x_n) \otimes a(x_{n+1}) < 0 \). Let us take as \( x_{n+1} \) the minimal compatible closure of \( x_n + x_{n+1} \) (it exists due to \( x_{n+1} \) being compatible).
with \( \text{OBJ}_P \). There are two possibilities: either we have constructed \( x_{\ell i} = x \), or \( x_{\ell i} < x \) (note that \( x_{\ell i} \leq x \) for any compatible closure \( F \) of \( x_{\ell i} + x \)), and, in particular, for \( x = x \).

The process of generating new \( x_{\ell i} \) cannot commence indefinitely since \( x < x_{\ell i} \leq x \), so \( x \) will eventually be constructed.

Avoiding unnecessary constraints One can see that the inequalities in the middle of (8) are not essential for an algorithm checking only compatible vectors. Indeed, they are just the result of the substitution of \( M = M_0 + C \cdot x \) into the constraints \( M > 0 \) and hold for any compatible vector \( x \) (see the proof of theorem 3). So these inequalities may be left out without losing any compatible solution. After eliminating these constraints, the reduced system

\[
\sum_{k \in K} \left( \sum_{j \in J} \alpha_j d_{k,j} - \sum_{j \in J} \alpha_j d_{k,j} \right) \leq \sum_{k \in K} \alpha_k - 1 - \sum_{k \in K} M_{\ell i}(k) \quad \text{for all } a \in F
\]

(10)
can have minimal solutions which are not compatible with \( \text{OBJ}_P \). However, such solutions are not computed by the algorithm as it checks only compatible vectors.

Sketch of the algorithm In the discussion below we refer to the parameters appearing in the generic system (6), since (10) is of that format.

The branching condition for (6) can be formulated as

\[
x_{\ell i} \text{ can be incremented by 1 if } \sum_{i=1}^{P} x_{\ell i} < 0,
\]

where each \( x_{\ell i} \) is given by

\[
x_{\ell i} = \begin{cases} 0 & \text{if } a_i \otimes x < d_i \text{ and } a_i \otimes x_j < 0 \\ (a_i \otimes x - d_i)(a_i \otimes x_j) & \text{otherwise}, \end{cases}
\]

where \( a_i = (a_{i1}, \ldots, a_{iP}) \). The algorithm in figure 7 starts from the tuple \((0, \ldots, 0)\), which is the root of the search tree and works in the way similar to the original CDA, but only checks vectors compatible with \( \text{OBJ}_P \).

In the general case, the maximal depth of the search tree is \( q \), so CDA needs to store \( q \) vectors of length \( g \). In our algorithm, we use just two arrays, \( X \) and \( \text{FIXED} \), of length \( q \).

- \( X \): array[1..q] of \( \{0, 1\} \)
  - To construct a solution.
- \( \text{FIXED} \): array[1..q] of \( \text{layers} \)
  - To keep an information about the level of fixing the components of \( X \).

The interpretation of these arrays is as follows:

- \( \text{FIXED}[k] = 0 \). Then \( X[k] \) must be 0 and this means that \( X[k] \) has not been considered yet, and may later be set to 1 or frozen.
- \( \text{FIXED}[k] = k > 0 \) and \( X[k] = 0 \). Then \( X[k] \) has been frozen at some node on level \( k \) whose subtree the algorithm is developing. It cannot be undone until the algorithm backtracks to the level \( k \).
- $P[X][d] = k > 0$ and $X[d] = 1$. Then $X[d]$ has been set to 1 at some node on level $k$ whose subtree the algorithm is developing. This value is fixed for the entire subtree.

Notice that storing the levels of fixing the elements of $X$ allows one to undo changes during backtracking, without keeping all the intermediate values of $X$.

We also use the following auxiliary variables and functions:

- $depth : \mathbb{R}$
  The current depth in the search tree.

- $freeze(k : \mathbb{R})$
  Freezes all $X[d]$ such that $a_k \leq x_m$. If there is $X[d] = 1$ to be frozen then $freeze$ fails. The corresponding elements of $P[X]$ are set to the current value of $depth$.

- $set(k : \mathbb{R})$
  Sets all $X[d]$ such that $a_k > x_m$ to 1 and uses $freeze$ to freeze all $X[d]$ such that $a_k < x_m$. If there is a frozen $X[d]$ to be set to 1, or $X[d] = 1$ to be frozen then $set$ fails. The current value of $depth$ is written in the elements of $P[X]$, corresponding to the components being fixed.

```
Input:
Cons — a system of constraints
Unf[0] — a finite complete prefix of the unfolding

Output:
A solution of Cons compatible with Unf[0], if it exists

Initialization:
depth <- 1
X <- (0, ..., 0)
For i \in \{1, ..., g\}: P[X][i] <- 1 if yi \in Pos
0 otherwise

Main procedure:
if $X$ is a solution of Cons then return $X$
for all $i \in \{k | 1 \leq k \leq g \land P[X][i] = 0 \land (1)$ holds$):
  depth <- depth + 1
if set($i$) has succeeded then
  apply the procedure to $X$ recursively
if solution is found then return $X$
undo changes in the elements $X[d]$ such that $P[X][d] = depth$
depth <- depth - 1
freeze($i$) never fails here as $X$ is compatible +
return solution not found
```

Fig. 7. Headlock checking algorithm

Further optimization: We now introduce some simple but useful heuristics. For example, if the algorithm has fixed some variables and found out that some of the inequalities have become impossible, then it may cut the current branch of the search graph. Moreover, we sometimes can determine the values of variables which have not yet been fixed,
or find out that some inequalities have become redundant. Let us consider the inequality
\[ a_1 x_1 + \cdots + a_n x_n \leq d \]
in the context of generating the search tree at the current node and calculate
\[ f_{ \text{min}} = \sum_{x_j \text{ is fixed}} a_j x_j, \quad \text{max} = \sum_{x_j \text{ is not fixed}} a_j x_j, \quad \text{mind} = \sum_{x_j \text{ is not fixed}} a_j x_j. \]

Note that since \( s \in (0,1) \), the \( \text{mind} \) and \( \text{max} \) are the bounds for the minimal and maximal possible values of the non-fixed part of the left hand side of the inequality in the subtree rooted in the current node, and one can show that:

- if \( \text{mind} > d \), \(-f_{\text{min}} \) then (12), and so the whole subtree, is infeasible and the current subtree of the search tree may be pruned;
- if \( \text{mind} \leq d \), \(-f_{\text{min}} \) then (12) is redundant and may be ignored in the subtree rooted at the current node of the search tree;
- if \( \text{mind} = d \), \(-f_{\text{min}} \) then (12) may only be satisfied if all its non-fixed variables with negative coefficients are equal to 1, and all its non-fixed variables with positive coefficients are equal to 0. After fixing these variables, (12) becomes redundant;
- if \( \text{mind} + a_k > d - f_{\text{min}} \) for a non-fixed variable \( x_k \) (in this case \( a_k < 0 \)), then (12) forces \( x_k \) to be 1;
- if \( \text{mind} - a_k < d - f_{\text{min}} \) for a non-fixed variable \( x_k \) (in this case \( a_k > 0 \)), then (12) forces \( x_k \) to be 0.

After fixing the value of a variable, it is necessary to build a compatible closure for the current vector; non-variable constraints can become fixed during this process, so the above tests are applied iteratively to all the inequalities in the system while it takes effect. If such a closure cannot be built, then the current subtree of the search tree does not contain a compatible solution and may be pruned.

The tests may also be applied before the algorithm starts; in such a case we may reduce the system of constraints by excluding redundant inequalities and substituting the values of fixed variables.

As an example, let us consider the following system of inequalities:
\[
\begin{align*}
-x_1 - 2x_2 + 2x_3 + 2x_4 + x_5 & \leq 1 \\
x_1 + 2x_2 & \leq 2 \\
2x_1 + 3x_2 + x_3 & \leq 3
\end{align*}
\]
where the variables \( x_1 \) and \( x_3 \) are fixed to 1 and \( x_4 \) is fixed to 0. Then, for the first inequality, \( f_{\text{min}} = -3 \), \( \text{mind} = 0 \) and \( \text{max} = 3 \), so it is redundant due to \( \text{max} \leq 1 - f_{\text{min}} \).

For the second inequality, \( f_{\text{min}} = 3 \), \( \text{mind} = -1 \) and \( \text{max} = 1 \); due to \( \text{max} = 2 - f_{\text{min}} \) we can determine that \( x_2 = 0 \) and \( x_3 = 1 \). For the third inequality, \( f_{\text{min}} = 3 \), \( \text{mind} = -3 \) and \( \text{max} = 3 \); it is possible to determine that \( x_1 = 1 \) (due to \( \text{mind} - (-3) > 3 - f_{\text{min}} \)) and \( x_3 = 0 \) (due to \( \text{mind} + 2 > 3 - f_{\text{min}} \)). After fixing this value, the inequality becomes redundant.

Suppose now that we added another constraint, \( x_1 + x_2 - 2x_3 + x_4 \leq 1 \), for which \( f_{\text{min}} = 2 \), \( \text{mind} = 0 \) and \( \text{max} = 1 \). Then the system becomes infeasible due to \( \text{mind} > 1 - f_{\text{min}} \).

Additional dependencies between events: The system of inequalities may contain additional information about causal dependencies which must hold in order to reach a
deadlock. For example, the effect of $x_k - x_j \leq 0$ is the same as if $x_j$ was a predecessor of $x_k$. Such an information can be incorporated directly into the unfolding before the algorithm starts.

Let us formulate simple tests for determining such dependencies. We observe that (12) implies $x_s \leq x_j$ if and only if $x_s \leq x_j$ implies that the inequality cannot be satisfied irrespective of the values of other variables. This happens if after fixing $x_s = 1$ and $x_j = 0$, (12) becomes infeasible, i.e., $\min > d' - f_{x_j}$. Thus, we can find all such pairs $(x_s, x_j)$ and then consider the following cases:

- $x_s = x_j$. Then we ignore this pair (the dependency is already represented in the unfolding).
- $x_s \leq x_j$. Then we fix $x_s = x_j$ for any execution sequence leading to a deadlock, so we can reduce the number of variables after the substitution $x_s = x_j$.
- $x_s \leq x_j$. Then we cannot occur in any execution sequence leading to a deadlock, and we may freeze $x_s$ and any other variable $x_k$ such that $x_s \leq x_k$.
- $x_s \geq x_j$. Then we may modify the unfolding by adding a new condition $b$ such that $b = \{x_j\}$ and $b' = \{x_s\}$.

Another kind of dependencies resembles that of the conflict relation. For example, the effect of $x_s + x_j \leq 1$ is similar to a conflict between $x_s$ and $x_j$ in the unfolding. The inequality (12) implies $x_s + x_j \leq 1$ if after fixing $x_s = 1$ and $x_j = 1$, it becomes infeasible, i.e., $\min > d' - f_{x_j}$. Thus we can find all such pairs $(x_s, x_j)$ and then consider the following cases:

- $x_s \leq x_j$. Then $x_j$ cannot occur in any execution sequence leading to a deadlock, so we may freeze $x_j$ and any other component $x_k$ such that $x_j \leq x_k$.
- $x_s \leq x_j$. Then, as before, we may freeze $x_s$ and any other component $x_k$ such that $x_j \leq x_k$.
- $x_s \leq x_j$. Then we ignore this pair (the dependency is already represented in the unfolding).
- $x_s \geq x_j$. Then we may modify the unfolding by adding a new initial condition $b$ such that $b = \{x_s\}$.

The dependencies discussed above may also emerge during a run of the algorithm due to some variables becoming fixed. In such a case, we do not modify the unfolding nor the system of constraints (otherwise we would have to undo changes during backtracking), but just for the values of some variables.

Shortest walk. Finding a shortest path leading to a deadlock may facilitate debugging. In such a case, we need to solve an optimisation problem with the same system of constraints as before, and $\mathcal{L}(x) = \sum x_i$ as a function to be minimised.

The algorithm can easily be adapted for this task. The only adjustment is not to stop after the first solution has been found, but to keep the current optimal solution together with the value of the function $\mathcal{L}$. As this function is non-decreasing, we may prune the branch of the search tree as soon as value of $\mathcal{L}$ becomes greater than, or equal to, the current optimal value. This also saves us from keeping all non-decomposable solutions found by the algorithm and used to prune the search tree as soon as the current node becomes greater than, or equal to, some non-decomposable solution (note that if $x' \leq x''$ then $\mathcal{L}(x') \leq \mathcal{L}(x'')$). It therefore 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 (in order to reduce the search space). This can be done by choosing in each step of the algorithm the 'most promising' branches. Since the ordering on the successors of a node in the ordered
version of CTA (see section 4) is arbitrary, we may exploit the information about the 
value of $c$ and check successors with smaller values first. Such an algorithm can be seen 
as a version of the “branch and bound” method which only considers vectors compatible 
with unfolding and uses frozen components and branching condition to reduce the search 
space.

## Experimental Results

For the experiments, we used the FPF tool [3] to generate finite complete preorders for 
our partial order algorithms, and for deadlock checking based on McMillan’s method 
([14, 15]) and the ATP algorithm4 [20]. The results in Table 1 have been measured on a 
PC with Pentium III 500MHz processor and 128MB RAM (building unfoldings for 
RW(5), SFM(11), FLAVATOR(3), and STACK(10) were aborted after 20 hours).

<table>
<thead>
<tr>
<th>Problem</th>
<th>Branding Preparations</th>
<th></th>
<th></th>
<th>Time (s)</th>
<th>RAM</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>l1d100</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>10.8</td>
<td>8.1</td>
<td>6.9</td>
</tr>
<tr>
<td>random</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6.2</td>
<td>5.9</td>
<td>5.8</td>
</tr>
<tr>
<td>l1d100</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>10.8</td>
<td>8.1</td>
<td>6.9</td>
</tr>
<tr>
<td>random</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6.2</td>
<td>5.9</td>
<td>5.8</td>
</tr>
<tr>
<td>l1d100</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>10.8</td>
<td>8.1</td>
<td>6.9</td>
</tr>
<tr>
<td>random</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6.2</td>
<td>5.9</td>
<td>5.8</td>
</tr>
<tr>
<td>l1d100</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>10.8</td>
<td>8.1</td>
<td>6.9</td>
</tr>
<tr>
<td>random</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6.2</td>
<td>5.9</td>
<td>5.8</td>
</tr>
<tr>
<td>l1d100</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>10.8</td>
<td>8.1</td>
<td>6.9</td>
</tr>
<tr>
<td>random</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6.2</td>
<td>5.9</td>
<td>5.8</td>
</tr>
<tr>
<td>l1d100</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>10.8</td>
<td>8.1</td>
<td>6.9</td>
</tr>
<tr>
<td>random</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6.2</td>
<td>5.9</td>
<td>5.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem</th>
<th>Branding Preparations</th>
<th></th>
<th></th>
<th>Time (s)</th>
<th>RAM</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>l1d100</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>10.8</td>
<td>8.1</td>
<td>6.9</td>
</tr>
<tr>
<td>random</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6.2</td>
<td>5.9</td>
<td>5.8</td>
</tr>
<tr>
<td>l1d100</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>10.8</td>
<td>8.1</td>
<td>6.9</td>
</tr>
<tr>
<td>random</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6.2</td>
<td>5.9</td>
<td>5.8</td>
</tr>
<tr>
<td>l1d100</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>10.8</td>
<td>8.1</td>
<td>6.9</td>
</tr>
<tr>
<td>random</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6.2</td>
<td>5.9</td>
<td>5.8</td>
</tr>
<tr>
<td>l1d100</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>10.8</td>
<td>8.1</td>
<td>6.9</td>
</tr>
<tr>
<td>random</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6.2</td>
<td>5.9</td>
<td>5.8</td>
</tr>
<tr>
<td>l1d100</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>10.8</td>
<td>8.1</td>
<td>6.9</td>
</tr>
<tr>
<td>random</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6.2</td>
<td>5.9</td>
<td>5.8</td>
</tr>
<tr>
<td>l1d100</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>10.8</td>
<td>8.1</td>
<td>6.9</td>
</tr>
<tr>
<td>random</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>6.2</td>
<td>5.9</td>
<td>5.8</td>
</tr>
</tbody>
</table>

### Table 1: Experimental Results

4 To solve the generated systems of constraints, the Ip solver general purpose LP-solver by 
M.H.C.M. Korfke was used.
Although our testing was limited in scope, it seems that the new algorithm is fast, even for large unfoldings. In [15], it has been pointed out that the HGP-approach is good for "wide" unfoldings with a high number of cut-off events, whereas for unfoldings with a small percentage of cut-off events, McMillan's approach is better. Our approach works well both for "wide" unfoldings with a high number of cut-off events and conflicts, and for "narrow" cases with a high number of causal dependencies. The worst case is the absence of both conflicts and partial order dependencies, i.e. when nearly all events are in the coalescence hierarchies. In this case, the number of events is dominated by the number of cut-off events. As the general problem is NP-complete in the size of unfolding, such examples can be artificially constructed, but we expect that the new algorithm should work well for practical verification problems.

7 Parallelisation aspects

The linear programming approach to deadlock detection described in this paper can be implemented in a set of parallel processing nodes. For a shared-memory architecture, we just unfold one step of the recursion and distribute the for all loop (see Algorithm 7) between processors, feeding some elements according to the frozen components rule. Each processor must have its own copy of the arrays X and Y.

The algorithm is also appropriate for a distributed memory architecture as the amount of message passing required is relatively low. In this case, each node must have its own copies of all arrays, the system of constraints, and the unfolding. If the algorithm is used for finding shortest path to a deadlock, then each computing node — as soon as it finds a solution which is better than the earlier ones — should broadcast the value of the function C in order that other nodes take this information into account and reduce their search space.

8 Other related model checking problems

The approach we presented may be generalised to other Petri net problems which can be reduced to the feasibility checking of a system of linear constraints. For example, we can check the mutual exclusion of two places a and a' in the net system E which amounts to saying that, for any reachable marking M, at least one of the places a and a' must be empty. In other words, M(a) ≥ 1 and M(a') ≥ 1 cannot hold simultaneously, so a sufficient condition for a and a' to be mutually exclusive is the infeasibility of the following system of linear constraints:

\[ \begin{align*}
M &= M_0 + M_x \\
M(a) &\geq 1 \quad \text{and} \quad M(a') \geq 1 \\
M &\in \mathbb{N}^x \quad \text{and} \quad x \in \mathbb{N}^a
\end{align*} \]

We can re-formulate this problem for \( \cup \mathcal{D}_E \), in the following way. The constraint \( M(a) \geq 1 \) (which means that a is marked in the original net system E) is equivalent to saying that at least one condition in \( \cup \mathcal{D}_E \) labelled by a is marked:

\[ \sum_{d \in \mathcal{D}_E(a)} M(d) \geq 1. \]

Footnote: A balanced distribution of tasks, it is better to create a queue of unprocessed recursive calls.
Hence the whole problem is equivalent to the infeasibility check of the following system of constraints:

\[
\begin{cases}
M = M_{oc} + C \cdot x \\
\sum_{b \in U^{-1}(a)} M(b) \geq 1 \quad \text{and} \quad \sum_{b \in U^{-1}(w)} M(b) \geq 1 \\
\pi(a) = 0 \text{ for all } a \in F_{out} \\
M \in \mathcal{M}^R \text{ and } x \in \mathcal{M}^F
\end{cases}
\]

Again, the variables \( M \) can be eliminated and constraints \( M \geq 0 \) omitted (they hold for any solution compatible with the unfolding — see section 3.2), so we need to check the infeasibility of the system:

\[
\begin{cases}
\sum_{b \in U^{-1}(a)} \left( \sum_{e \in u} \pi(e) - \sum_{w \in v} \pi(w) \right) \geq 1 - \sum_{b \in U^{-1}(a)} M_{oc}(b) \\
\sum_{b \in U^{-1}(w)} \left( \sum_{e \in u} \pi(e) - \sum_{w \in v} \pi(w) \right) \geq 1 - \sum_{b \in U^{-1}(w)} M_{oc}(b) \\
\pi(a) = 0 \text{ for all } a \in F_{out} \\
M \in \mathcal{M}^R \text{ and } x \in \mathcal{M}^F
\end{cases}
\]

and the algorithm proposed in this paper can be used for this purpose.

Another interesting problem is marking reachability (and coverability). Let \( M' \) be a marking of \( \Sigma \). Then the system of constraints has the form:

\[
\begin{cases}
M = M_{oc} + C \cdot x \\
\sum_{b \in U^{-1}(a)} M(b) = (\geq) M'(a) \quad \text{for all } a \in S \\
\pi(a) = 0 \text{ for all } a \in F_{out} \\
M \in \mathcal{M}^R \text{ and } x \in \mathcal{M}^F
\end{cases}
\]

and, after eliminating the variables \( M \) and constraints \( M \geq 0 \), we need to check the feasibility of the following system:

\[
\begin{cases}
\sum_{b \in U^{-1}(a)} \left( \sum_{f \in u} \pi(f) - \sum_{w \in v} \pi(w) \right) = (\geq) M'(a) - \sum_{b \in U^{-1}(a)} M_{oc}(b) \quad \text{for all } a \in S \\
\pi(a) = 0 \text{ for all } a \in F_{out}
\end{cases}
\]

The algorithm can be easily adapted to solve this system, and if it has a compatible solution then the marking \( M' \) is reachable (coverable), and any compatible solution gives an execution sequence leading to it (respectively, to a marking which covers it, otherwise \( M' \) is unreachable (respectively, uncoverable).

9 Conclusions

Experiments indicate that the algorithm we proposed in this paper can solve problems with thousands of variables. This overcomes the existing limitations, as MCF-problems...
with even a few hundreds of integer variables are usually a hard task for general purpose
solvers. It is worth emphasizing that the limitation was not the size of computer memory,
but rather the time to solve an NP-complete problem. With our approach, the main
limitation becomes the size of memory to store the unfolding. Our future research will
aim at developing effective parallel algorithms (especially for a distributed memory
architecture) for constructing large unfoldings which cannot fit in the memory of a single
processing node, and modifying our algorithm to handle such 'distributed' unfoldings.
Finally, our experiments have indicated that the process of model checking a finite
complete prefix is often faster than the process of constructing such a prefix. We therefore
plan to investigate novel algorithms for fast generation of net unfoldings.

Finally, it is worth pointing out that the approach presented in this paper is also
valid for Petri Nets with weighted arcs since all the arcs in their unfoldings are unitary.

Acknowledgements
We would like to thank Paul Watson for his support and comments on an earlier version
of this paper, and Christian Stokes for his help with using the PEP tool. The first author
was supported by an ORS Awards Scheme grant ORS/CR20/4 and by an EPSRC grant
GR/M99986.

References
1. P. Ajili and H. Camesasca: Complete Solving of Linear Diophantine Equations and Inequal-
ities Without Adding Variables. Proc. of the International Conference on principles and
2. P. Ajili and H. Camesasca: Avoiding Slack Variables in the Solving of Linear Diophantine
3. H. Heil and R. Gehr (1994) — more than a Petri Net! Proc. of TACAS94—Tools and
4. R. Clarke, E. Emerson and A. Sistla: Automatic Verification of Finite-state Con-
5. H. Camesasca: Solving Linear Diophantine Constraints Incrementally. Proc. of 9th Inter-
7. H. Camesasca and H. Devic: An Efficient Incremental Algorithm for Solving Systems of Lin-
(1994).
Proc. of TACAS94—Tools and Algorithms for the Construction and Analysis of Systems,
Margra, T., Sittenh H. (Ed.). Springer-Verlag, Lecture Notes in Computer Science 1000
13. S. Krivok: About Some Methods of Solving and Possibility Criteria of Linear Diophantine
Equations over Natural Numbers Domain (In Russian). Cybernetics and System Analysis 4
14. K. L. McMillan: Using Unfoldings to Avoid State Explosion Problem in the Verification of
Asynchronous Circuits. Proc. of 4th Workshop on Computer Aided Verification, Springer-
