LP Deadlock Checking Using Partial Order Dependencies

Victor Khomenko and Maciej Koutny

Department of Computing Science, University of Newcastle
Newcastle upon Tyne NE1 7RU, U.K.
{Victor.Khomenko, Maciej.Koutny}@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 nets. In [15], 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 Contejean-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. 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.

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 model checking [4]. The main drawback of model checking is that it suffers from the state 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 ([10, 11, 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 net unfolding technique presented in [14, 15] reduces memory requirement, but the deadlock checking algorithms proposed are quite slow, even for medium-size unfoldings.

In [15], the problem of deadlock checking a Petri net was reduced to a mixed integer linear programming problem. In this paper, we present a further development of this approach. We adopt the Contejean-Devie's algorithm ([1, 2, 5–7]) 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 conflicts 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, 5–7] and recalls the main aspects of the Contejean-Devie's Algorithm (CDA) for solving systems of linear constraints over the natural numbers 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, useful heuristics, as well as outlining 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, while section 7 describes possible directions for future research. The proofs of the results can be found in [12].

2 Basic definitions

In this section, we first present basic definitions concerning Petri nets, and then recall (see [10]) notions related to net unfoldings.

Petri nets 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. $M : S \to \mathbb{N} = \{0, 1, 2, \ldots\}$. As usual, we will denote ${}^\bullet z = \{y \mid (y, z) \in F\}$ and $z^\bullet = \{y \mid (z, y) \in F\}$, for all $z \in S \cup T$. We will assume that ${}^\bullet t \neq \emptyset \neq t^\bullet$, for every $t \in T$.

A *net system* is a pair $\Sigma = (N, M_0)$ comprising a finite net $N = (S, T, F)$ and an (initial) marking M_0. A transition $t \in T$ is *enabled* at a marking M, denoted $M[t\rangle$, if for every $s \in {}^\bullet t$, $M(s) \geq 1$. Such a transition can be *executed*, leading to a marking M' defined by $M' = M - {}^\bullet t + t^\bullet$. We denote this by $M[t\rangle M'$ or

$M[\rangle M'$. The set of *reachable* markings of Σ is the smallest (w.r.t. set inclusion) set $[M_0\rangle$ containing M_0 and such that if $M \in [M_0\rangle$ and $M[\rangle M'$ then $M' \in [M_0\rangle$. For a finite sequence of transitions, $\sigma = t_1 \dots t_k$, we denote $M_0[\sigma\rangle M$ if there are markings $M_1, \dots, M_k$ such that $M_k = M$ and $M_{i-1}[t_i\rangle M_i$, for $i = 1, \dots, k$.

A marking is *deadlocked* if it does not enable any transitions. The net system Σ 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 $M(S) \subseteq \{0, \dots, k\}$, for every reachable marking M.

Marking equation Let $\Sigma = (N, M_0)$ be a net system, and $S = \{s_1, \dots, s_m\}$ and $T = \{t_1, \dots, t_n\}$ be sets of its places and transitions, respectively. We will often identify a marking M of Σ with a vector $M = (\mu_1, \dots, \mu_m)$ 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 $j \leq n$,

$$\mathcal{N}_{ij} = \begin{cases} 1 & \text{if } s_i \in t_j^\bullet \setminus {}^\bullet t_j \\ -1 & \text{if } s_i \in {}^\bullet t_j \setminus t_j^\bullet \\ 0 & \text{otherwise .} \end{cases}$$

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

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

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 $y\#y'$, if there are distinct transitions $t, t' \in T$ such that ${}^\bullet t \cap {}^\bullet t' \neq \emptyset$ and (t, y) and (t', y') are in the reflexive transitive closure of the flow relation F, denoted by $\preceq$. A node y is in *self-conflict* if $y\#y$.

An *occurrence net* is a net $ON = (B, E, G)$ where B is the set of *conditions* (places) and E is the set of *events* (transitions). It is assumed that: ON is acyclic (i.e. $\preceq$ is a partial order); for every $b \in B$, $|{}^\bullet b| \leq 1$; for every $y \in B \cup E$, $\neg(y\#y)$ and there are finitely many y' such that $y' \prec y$, where $\prec$ denotes the irreflexive transitive closure of G. $\mathit{Min}(ON)$ will denote the minimal elements of $B \cup E$ with respect to $\preceq$. The relation $\prec$ is the *causality relation.* Two nodes are *co-related*, denoted by $y \ co \ y'$, if neither $y\#y'$ nor $y \preceq y'$ nor $y' \preceq y$.

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

A *branching process* of Σ [9] is a quadruple $\pi = (B, E, G, h)$ such that (B, E, G) is an occurrence net and h is a homomorphism from ON to Σ. A branching process $\pi' = (B', E', G', h')$ of Σ is a *prefix* of a branching process $\pi = (B, E, G, h)$, denoted by $\pi' \sqsubseteq \pi$, if (B', E', G') is a subnet of (B, E, G) such that: if $e \in E'$ and $(b, e) \in G$ or $(e, b) \in G$ then $b \in B'$; if $b \in B'$ and $(e, b) \in G$ then $e \in E'$; and h' is the restriction of h to $B' \cup E'$. For each Σ there exists a unique (up to isomorphism) maximal (w.r.t. $\sqsubseteq$) branching process, called the *unfolding* of Σ.

Configurations and cuts A *configuration* of an occurrence net ON is a set of events C such that for all $e, f \in C$, $\neg(e \# f)$ and, for every $e \in C$, $f \prec e$ implies $f \in C$. A *cut* is a maximal w.r.t. set inclusion set of conditions B' such that $b \mathrel{co} b'$, for all $b, b' \in B'$. Every marking reachable from $\mathit{Min}(ON)$ is a cut.

Let C be a finite configuration of a branching process π. Then $\mathit{Cut}(C) = (\mathit{Min}(ON) \cup C^\bullet) \setminus {}^\bullet C$ is a cut; moreover, the multiset of places $h(\mathit{Cut}(C))$ is a reachable marking of Σ, denoted $\mathit{Mark}(C)$. A marking M of Σ is *represented* in π if the latter contains a finite configuration C such that $M = \mathit{Mark}(C)$. Every marking represented in π is reachable, and every reachable marking is represented in the unfolding of Σ.

A branching process π of Σ is *complete* if for every reachable marking M of Σ, there is a configuration C in π such that $\mathit{Mark}(C) = M$, and for every transition t enabled by M, there is a configuration $C \cup \{e\}$ such that $e \notin C$ and $h(e) = t$.

Although, in general, an unfolding is infinite, for every bounded net system Σ one can construct a finite complete prefix Unf_Σ of the unfolding of Σ. Moreover, there are *cut-off* events[1] in Unf_Σ such that, for every reachable marking M of Σ, there exists a configuration C in Unf_Σ such that $M = \mathit{Mark}(C)$ 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 $\mathit{Unf}_\Sigma = (B, E, G, h)$ is a finite complete prefix of the unfolding of a bounded net system $\Sigma = (S, T, F, M_0)$. We will denote by M_{in} the canonical initial marking of Unf_Σ 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 $e_1, e_2, \ldots, e_q$ are respectively the conditions and events of Unf_Σ, and that $\mathcal{C}$ is the $p \times q$ incidence matrix of Unf_Σ. The set of cut-off events of Unf_Σ will be denoted by E_{cut}.

We now recall the main results from [15]. A finite and complete prefix Unf_Σ may be treated as an acyclic safe net system with the initial marking M_{in}. Each

[1] Intuitively, cut-off events are nodes at which the potentially infinite unfolding may be cut without losing any essential information about the behaviour of Σ; see [9–11, 14, 15] for details.

reachable deadlocked marking in Σ is represented by a deadlocked marking in Unf_Σ. However, some deadlocked markings of Unf_Σ lie beyond the cut-off events and may not correspond to deadlocks in Σ. 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. *([15]) Σ is deadlock-free if and only if the following system has no solution (in M and x):*

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

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}, it is possible to require that x be a binary vector, $x \in \{0,1\}^q$.

To solve the resulting mixed-integer LP-problem, [15] 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 x corresponding to cut-off events are set to 0.

4 Solving systems of linear constraints

In this paper, we will adapt the approach proposed in [1, 2, 5–7], in order to solve Petri net problems which can be reformulated as LP-problems. We start by recalling some basic results.

The original *Contejean and Devie's algorithm (CDA)* [5–7] solves a system of linear homogeneous equations with arbitrary integer coefficients

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

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

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

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

$$a(x) = \begin{pmatrix} a_{11}x_1 + \cdots + a_{1q}x_q \\ a_{21}x_1 + \cdots + a_{2q}x_q \\ \vdots \qquad\qquad \vdots \\ a_{p1}x_1 + \cdots + a_{pq}x_q \end{pmatrix} = x_1 \cdot a(\varepsilon_1) + \cdots + x_q \cdot a(\varepsilon_q)\ , \tag{3}$$

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 (2) can be represented by a finite basis $\mathcal{B}$ which is the minimal (w.r.t. set inclusion) subset of $\mathcal{S}$ such that every solution is a linear combination with non-negative integer coefficients 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 $i \leq q$; moreover, $x < x'$ 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 $\mathbf{0}$. 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.

> 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 figure 1).

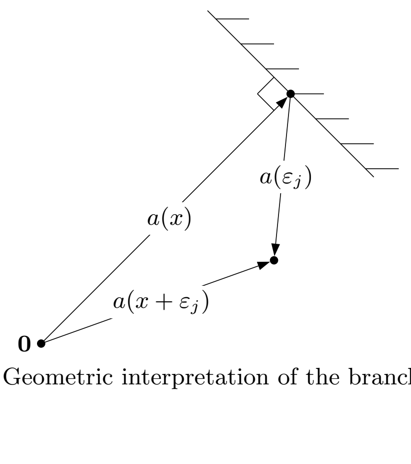

Fig. 1. Geometric interpretation of the branching condition in CDA

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 $x = (x_1, \dots, x_q)$,

$$\text{increment by 1 an } x_j \text{ satisfying } a(x) \odot a(\varepsilon_j) < 0 \;, \tag{4}$$

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. *([7]) The following hold for the CDA shown in figure 2:*

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

- search breadth-first a directed acyclic graph rooted at $\varepsilon_1, \dots, \varepsilon_q$
- **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. 2. CDA (breath-first version)

But this algorithm may perform redundant calculations as some vectors can be computed more than once. This can be remedied by using *frozen components*, defined thus. Assume that there is a total ordering $\prec_x$ on the sons of each node x of the search graph constructed by CDA.

> 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 i-th component is *frozen* in the sub-graph rooted at $x + \varepsilon_j$ 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.

The ordered version of CDA can be extended to handle bounds imposed on variables and homogeneous systems of equations and inequalities (see [1, 2, 5–7]). Moreover, it can be adapted to solve non-homogeneous systems of inequalities

$$\begin{cases} a_{11}x_1 + \cdots + a_{1q}x_q \leq d_1 \\ a_{21}x_1 + \cdots + a_{2q}x_q \leq d_2 \\ \quad \vdots \qquad\qquad\quad \vdots \qquad \vdots \\ a_{p1}x_1 + \cdots + a_{pq}x_q \leq d_p \end{cases} \tag{5}$$

5 An algorithm for deadlock detection

The MIP problem obtained in section 3 can be reduced to a pure integer one, by substituting the expression for M given by the marking equation into other constraints and, at the same time, reducing the total number of constraints. Each equation in $M = M_{in} + \mathcal{C} \cdot x$ has the form

$$M(b) = M_{in}(b) + \sum_{f \in {}^\bullet b} x(f) - \sum_{f \in b^\bullet} x(f) \qquad \text{for } b \in B. \tag{6}$$

After substituting these into (1) we obtain the system

$$\begin{cases} \sum_{b \in {}^\bullet e} \left(\sum_{f \in {}^\bullet b} x(f) - \sum_{f \in b^\bullet} x(f) \right) \le |{}^\bullet e| - 1 - \sum_{b \in {}^\bullet e} M_{in}(b) & \text{for all } e \in E \\ M_{in}(b) + \sum_{f \in {}^\bullet b} x(f) - \sum_{f \in b^\bullet} x(f) \ge 0 & \text{for all } b \in B \\ x \in \{0,1\}^q \text{ and } x(e) = 0 \text{ for all } e \in E_{cut} & \end{cases} \tag{7}$$

As (7) is a pure integer problem, CDA is directly applicable. However, since the number of variables can be large, it needs further refinement.

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 Unf_Σ. For example, if we set $x(e) = 1$ then each $x(f)$ such that f is a predecessor (in causal order) of e must be equal to 1, and each $x(g)$ such that g is in conflict with e, must be equal to 0. Similarly, if we set $x(e) = 0$ then no event f for which e is a cause can be executed in the same run, and so $x(f)$ 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 heuristics, can lead to significant speedups.

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

$$f \prec e \Rightarrow x(f) = 1 \quad \textit{and} \quad f \# e \Rightarrow x(f) = 0\ .$$

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

Theorem 3. *A vector $x \in \{0,1\}^q$ is compatible with Unf_Σ if and only if there exists an execution sequence starting at M_{in} whose Parikh vector is x.*

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

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 x (see the definition below) in each step of CDA and freezing all $x(e)$ such that $e \in E_{cut}$.

Definition 2. *A vector* $\overline{x} \in \{0,1\}^q$ *is a* compatible closure *of* $x \in \{0,1\}^q$ *if* $x \leq \overline{x}$ *and* $\overline{x}$ *is compatible with* Unf_{Σ}*. Moreover,* $\overline{x}$ *is a* minimal *compatible closure if it is minimal with respect to* $\leq$ *among all possible compatible closures of* x*.*

Let us consider the causal ordering $e_1 \prec e_2 \prec e_3$, $e_2 \prec e_4$ and e_3 *co* e_4 (see figure 3), and $x = (1,0,1,0)$. Then $\overline{x}' = (1,1,1,0)$ and $\overline{x}'' = (1,1,1,1)$ are compatible closures of x, and $\overline{x}'$ is the minimal one.

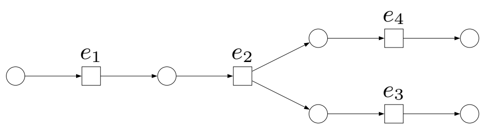

Fig. 3. An occurrence net

Theorem 4. *A vector* $x \in \{0,1\}^q$ *has a compatible closure if and only if for all* $e, f \in E$*,* $x(e) = x(f) = 1$ *implies* $\neg(e\#f)$*. If* x *has a compatible closure then its minimal 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 its minimal compatible closure.*

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 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 3, 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 $x_1 = 1$. Then we start the algorithm 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 $a(x) \odot a(\varepsilon_2 + \varepsilon_4) = 2 \not< 0$, so we 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 we choose only the first element ε_{i_1} for which $a(\varepsilon_{i_1})$ does return to the origin, and then add the remaining ones in order for $x + \varepsilon_{i_1} + \cdots + \varepsilon_{i_k}$ to be compatible, without worrying where they actually lead, as it shown in figure 4 (if there is no compatible closure of $x + \varepsilon_{i_1}$ then ε_{i_1} cannot be chosen). This means that we check the condition $a(x) \odot a(\varepsilon_{i_1}) < 0$ which coincides with the original CDA's branching condition, though we are moving along possibly different vector. As the following theorem shows, this technique is still complete.

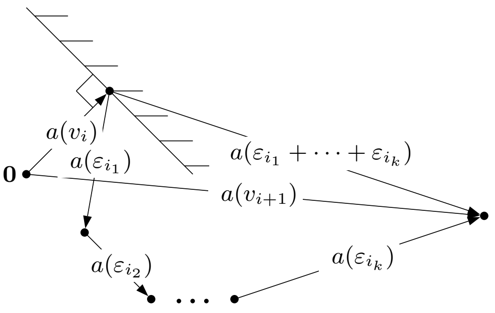

Fig. 4. Geometric interpretation of the new branching condition ($a(\varepsilon_{i_1})$ is 'returning to the origin' although $a(\varepsilon_{i_1} + \cdots + \varepsilon_{i_k})$ may not posses this property; here $v_{i+1} = v_i + \varepsilon_{i_1} + \cdots + \varepsilon_{i_k}$ is the minimal compatible closure of $v_i + \varepsilon_{i_1}$)

Theorem 5. *Every non-trivial minimal compatible solution can be computed using the above method.*

One can prove that the inequalities in the middle of (7) are not essential for an algorithm checking only compatible vectors. Indeed, they are just the result of the substitution of $M = M_{in} + \mathcal{C} \cdot x$ into the constraints $M \geq \mathbf{0}$ and hold for any compatible vector x (see the proof of theorem 3 in [12]). Hence these inequalities can be left out without adding any compatible solution. The reduced system

$$\begin{cases} \sum_{b \in {}^\bullet e} \left(\sum_{f \in {}^\bullet b} x(f) - \sum_{f \in b^\bullet} x(f) \right) \leq |{}^\bullet e| - 1 - \sum_{b \in {}^\bullet e} M_{in}(b) \quad \text{for all } e \in E \\ x \in \{0,1\}^q \text{ and } x(e) = 0 \text{ for all } e \in E_{cut} \end{cases} \tag{8}$$

can have minimal solutions which are not compatible with Unf_Σ; however, such solutions are not computed by an algorithm checking only compatible vectors.

Sketch of the algorithm In the discussion below we refer to the parameters appearing in the generic system (5), since (8) is of that format. The branching condition for (5) can be formulated as

$$x_j \text{ can be incremented by 1 if } \sum_{i=1}^{p} r_i < 0 \;, \tag{9}$$

where each r_i is given by

$$r_i = \begin{cases} 0 & \text{if } a_i \odot x < d_i \text{ and } a_i \odot \varepsilon_j < 0 \\ (a_i \odot x - d_i)(a_i \odot \varepsilon_j) & \text{otherwise ,} \end{cases}$$

where $a_i{=}(a_{i1}, \ldots, a_{iq})$. The algorithm in figure 5 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 Unf_{Σ}.

In the general case, the maximal depth of the search tree is q, so CDA needs to store q vectors of length q. In our algorithm, we use just two arrays, X and *FIXED*, of length q:

– $X : array[1..q]\ of\ \{0, 1\}$
  To construct a solution.
– $FIXED : array[1..q]\ of\ integers$
  To keep an information about the levels of fixing the components of X.

The interpretation of these arrays is as follows:

– $FIXED[i] = 0$. Then $X[i]$ must be 0 and this means that $X[i]$ has not been considered yet, and may later be set to 1 or frozen.
– $FIXED[i] = k > 0$ and $X[i] = 0$. Then $X[i]$ has been frozen at some node on level k whose subtree the algorithm is developing. It cannot be unfrozen until the algorithm backtracks to the level k.
– $FIXED[i] = k > 0$ and $X[i] = 1$. Then $X[i]$ has been set to 1 at some node on level k whose subtree the algorithm is developing. This value is fixed for the entire subtree.

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

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

Further optimisation Various heuristics used by general purpose MIP-solvers can be implemented to prune the search tree. For example, if the algorithm has fixed some variables and found out that some of the inequalities have become

```
Input
    $Cons$ — a system of constraints
    $Unf_{\Sigma}$ — a finite complete prefix of the unfolding

Output
    A solution of $Cons$ compatible with $Unf_{\Sigma}$ if it exists

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

Main procedure
    if $X$ is a solution of $Cons$ then return $X$
    for all $i \in \{k \mid 1 \leq k \leq q \wedge FIXED[k] = 0 \wedge (9) \text{ holds}\}$
        $depth \leftarrow depth + 1$
        if $set(i)$ has succeeded then
            apply the procedure to $X$ recursively
            if solution is found then return $X$
        undo changes in the elements $X[j]$ such that $FIXED[j] = depth$
        $depth \leftarrow depth - 1$
        $freeze(i)$ /* never fails here as $X$ is compatible */
    return solution not found
```

Fig. 5. Deadlock checking algorithm

infeasible, 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 (see [12] for more details).

After fixing the value of a variable, it is necessary to build a compatible closure for the current vector; new variables can become fixed, so the process can be applied iteratively 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.

Shortest trail 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 $\mathcal{L}(x) = x_1 + \cdots + x_q$ as a function to be minimised.

The algorithm can easily be adopted 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 corresponding value of the function $\mathcal{L}$ (see [12] for more details).

Problem	Deadlock-free	Original net $\lvert S\rvert$	$\lvert T\rvert$	Unfolding $\lvert B\rvert$	$\lvert E\rvert$	$\lvert E_{cut}\rvert$	Time [s] McM	MIP	PO
buf100	✓	200	101	10101	5051	1	0.01	24577	0.02
mutual	✓	49	41	887	479	79	4.42	70	0.02
ab_gesc	✓	58	56	3326	1200	511	33.93	260	0.17
sdl_arg	✓	163	96	644	199	10	0.04	20	<0.01
sdl_arg_ddlk		157	92	657	223	7	0.01	25	<0.01
RW(2)	✓	54	60	498	147	53	0.02	1	<0.01
RW(3)	✓	72	120	4668	1281	637	22.14	time	0.06
RW(4)	✓	94	224	51040	13513	7841	mem	—	14.55
SEM(2)	✓	27	25	61	32	5	<0.01	<0.01	<0.01
SEM(3)	✓	38	36	165	86	17	<0.01	1	<0.01
SEM(4)	✓	49	47	417	216	49	0.09	5	<0.01
SEM(5)	✓	60	58	1013	522	129	2.30	81	0.02
SEM(6)	✓	71	69	2393	1228	321	37.33	time	0.16
SEM(7)	✓	82	80	5533	2830	769	531.50	—	1.17
SEM(8)	✓	93	91	12577	6416	1793	time	—	8.75
SEM(9)	✓	104	102	28197	14354	4097	—	—	70.25
SEM(10)	✓	115	113	62505	31764	9217	—	—	585.13
ELEVATOR(1)	✓	59	72	518	287	9	0.10	27	<0.01
ELEVATOR(2)	✓	83	130	29413	15366	1796	mem	time	38.50
STACK(3)		20	24	320	174	26	<0.01	3	<0.01
STACK(4)		24	30	968	525	80	0.08	79	<0.01
STACK(5)		28	36	2912	1578	242	4.28	2408	0.01
STACK(6)		32	42	8744	4737	728	145.01	time	0.05
STACK(7)		36	48	26240	14214	2186	mem	—	0.16
STACK(8)		40	54	78728	42645	6560	—	—	0.52
STACK(9)		44	60	236192	127938	19682	—	—	7.36

buf100	—	buffer with 2^{100} states
mutual	—	mutual exclusion algorithm
ab_gesc	—	alternating bit protocol
sdl_arg	—	automatic request protocol
sdl_arg_ddlk	—	automatic request protocol (with deadlock)
RW(n)	—	reader-writer with n readers
SEM(n)	—	semaphore example with n processes
ELEVATOR(n)	—	example with n elevators
STACK(n)	—	stack of the depth n with test for fullness
time	—	the test had not stopped after 15 hours
mem	—	the test terminated because of memory overflow

Table 1. Experimental results

Parallelisation aspects The linear programming approach to deadlock detection described in this paper can easily be implemented on 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 figure 5) between processors[2], freezing some elements according to the frozen components rule. Each processor must have its own copy of the arrays X and FIXED.

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.

[2] For a balanced distribution of tasks, it is better to create a queue of unprocessed recursive calls.

6 Experimental results

For the experiments, we used the PEP tool [3] to generate finite complete prefixes for our partial order algorithm, and for deadlock checking based on McMillan's method ([14, 15]) and the *MIP* algorithm[3] ([15]). The results in table 1 have been measured on a PC with *Pentium*^{TM} III/500MHz processor and 128M RAM (building unfoldings for RW(5), SEM(11), ELEVATOR(3), and STACK(10) were aborted after 20 hours).

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 *MIP*-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. It appears that our approach works well both for 'wide' unfoldings 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 the absence of both conflicts and partial order dependencies (i.e. when nearly all events are in the *co* relation) combined with a small percentage 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 Conclusions

Experiments indicate that the algorithm we propose in this paper can solve problems with thousands of variables. This overcomes the existing limitations, as *MIP*-problems with even a few hundreds of integer variables are usually a hard task for general purpose solvers. It is worth emphasising 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 an effective parallel algorithm (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. 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.

Another possible direction is to investigate an obvious generalisation of the algorithm to systems of non-linear constraints. In this case we cannot use the pruning condition described in this paper to reduce the search space, but still need to check only compatible vectors.

Finally, [12] discusses how the approach presented here can be generalised to deal with other relevant verification problems, such as mutual exclusion, coverability and reachability analysis.

[3] To solve the generated system of constraints, the `lp_solve` general purpose LP-solver by M.R.C.M. Berkelaar was used.

Acknowledgements We would like to thank Paul Watson for his support and comments on an earlier version of this paper, Alexander Letichevsky and Sergei Krivoi for drawing our attention to the Contejean-Devie's algorithm and discussions on the theory of Diophantine equations, and Christian Stehno for his help with using the PEP tool. We would also like to thank the anonymous referees for their helpful comments. This research was supported by an ORS Awards Scheme grant ORS/C20/4 and by an EPSRC grant GR/M99293.

References

1. F. Ajili and E. Contejean: Complete Solving of Linear Diophantine Equations and Inequations Without Adding Variables. Proc. of *1st International Conference on principles and practice of Constraint Programming*, Cassis (1995) 1–17.
2. F. Ajili and E. Contejean: Avoiding Slack Variables in the Solving of Linear Diophantine Equations and Inequations. *Theoretical Comp. Sci.* 173 (1997) 183–208.
3. E. Best and B. Grahlmann: PEP — more than a Petri Net Tool. Proc. of *TACAS'96: Tools and Algorithms for the Construction and Analysis of Systems*, Margaria T., Steffen B. (Eds.). Springer-Verlag, Lecture Notes in Computer Science 1055 (1996) 397–401.
4. E. M. Clarke, E. A. Emerson and A. P. Sistla: Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. *ACM TOPLAS* 8 (1986) 244–263.
5. E. Contejean: Solving Linear Diophantine Constraints Incrementally. Proc. of *10th Int. Conf. on Logic Programming*, D. S. Warren (Ed.). MIT Press (1993) 532–549.
6. E. Contejean and H. Devie: Solving Systems of Linear Diophantine Equations. Proc. of *3rd Workshop on Unification*, University of Keiserlautern (1989).
7. E. Contejean and H. Devie: An Efficient Incremental Algorithm for Solving Systems of Linear Diophantine Equations. *Inf. and Computation* 113 (1994) 143–172.
8. CPLEX Corporation: *CPLEX 3.0*. Manual (1995).
9. J. Engelfriet: Branching processes of Petri Nets. *Acta Inf.* 28 (1991) 575–591.
10. J. Esparza, S. Römer and W. Vogler: An Improvement of McMillan's Unfolding Algorithm. Proc. of *TACAS'96: Tools and Algorithms for the Construction and Analysis of Systems*, Margaria T., Steffen B. (Eds.). Springer-Verlag, Lecture Notes in Computer Science 1055 (1996) 87–106.
11. J. Esparza: Model Checking Based on Branching Processes. *Science of Computer Programming* 23 (1994) 151–195.
12. V. Khomenko and M. Koutny: Deadlock Checking Using Liner Programming and Partial Order Dependencies. Teachnical Report CS-TR-695, Department of Computing Science, University of Newcastle (2000).
13. S. Krivoi: About Some Methods of Solving and Feasibility Criteria of Linear Diophantine Equations over Natural Numbers Domain (in Russian). *Cybernetics and System Analysis* 4 (1999) 12–36.
14. K. L. McMillan: Using Unfoldings to Avoid State Explosion Problem in the Verification of Asynchronous Circuits. Proc. of *CAV'92*, Springer-Verlag, Lecture Notes in Computer Science 663 (1992) 164–174.
15. S. Melzer and S. Römer: Deadlock Checking Using Net Unfoldings. Proc. of *CAV'97*, O. Grumberg (Ed.). Springer-Verlag, Lecture Notes in Computer Science 1254 (1997) 352–363.