Applying CEGAR to the Petri Net State Equation

Harro Wimmel and Karsten Wolf

Universität Rostock, Institut für Informatik

Abstract. We propose a reachability verification technique that combines the *Petri net state equation* (a linear algebraic overapproximation of the set of reachable states) with the concept of *counterexample guided abstraction refinement*. In essence, we replace the search through the set of reachable states by a search through the space of solutions of the state equation. We demonstrate the excellent performance of the technique on several real-world examples. The technique is particularly useful in those cases where the reachability query yields a negative result: While state space based techniques need to fully expand the state space in this case, our technique often terminates promptly. In addition, we can derive some diagnostic information in case of unreachability while state space methods can only provide witness paths in the case of reachability.

Keywords: Petri Net, Reachability Problem, Integer Programming, CEGAR, Structure Analysis, Partial Order Reduction.

1 Introduction

Reachability is *the* fundamental verification problem. For place/transition Petri nets (which may have infinitely many states), it is one of the hardest decision problems known among the naturally emerging yet decidable problems in computer science. General solutions have been found by Mayr [12] and Kosaraju [7] with later simplifications made by Lambert [9], but there are complexity issues. All these approaches use coverability graphs which can have a non-primitive-recursive size with respect to the corresponding Petri net. A new approach by Leroux [10] not using such graphs gives some hope, but a concrete upper bound for the worst case complexity so far eludes us. In a sense even worse, Lipton [11] has shown that the problem is **EXPSpace**-hard, so any try at programming a tool efficiently solving this problem to the full extent must surely fail.

Nevertheless, efficient tools exist that are applicable to a considerable number of problem instances. Model checkers, symbolic [2] or with partial order reduction [17], have been used successfully to solve quite large reachability problems. On a positive answer, a model checker can typically generate a trace, i.e. a firing sequence leading to the final marking. In contrast, negative answers are usually not accompanied by any diagnostic information. Such information, i.e. a counterexample or reasoning why the problem has a negative solution would require a deep analysis of the structure of the Petri net. So far, no tools are known that analyze the structure of a net and allow for such reasoning.

This paper presents an approach to the reachability problem that combines two existing methods. First, we employ the *state equation* for Petri nets. This is a linear-algebraic

overapproximation on the set of reachable states. Second, we use the concept of *counterexample guided abstraction refinement* (CEGAR) [3] for enhancing the expressiveness of the state equation. In essence, we iteratively analyse spurious solutions of the state equation and add constraints that exclude a solution found to be spurious but do not exclude any real solution. The approach has several advantages compared to (explicit or symbolic) purely state space based verification techniques:

- The search is quite focussed from the beginning as we traverse the solution space of the state equation rather than the set of reachable states;
- The search is close to breadth-first traversal, so small witness traces are generated;
- The method may perform well on unreachable problem instances (where state space techniques compute maximum size state spaces);
- In several unreachable problem instances, some kind of diagnostic information can be provided;
- A considerable workload can be shifted to very mature tools for solving linear programming problems.

In Sect. 2 we give the basic definitions. Section 3 shows how to use integer programming tools to find candidates for a solution. Section 4 deals with the analysis of the Petri net structure that is needed to push the integer programming onto the right path. In Sect. 5 we use methods of partial order reduction to mold the results of the integer programming into firing sequences solving the reachability problem. Finally, section 6 compares the results of an implementation with another model checker, showing that structure analysis can compete with other approaches.

2 The Reachability Problem

Definition 1 (Petri net, marking, firing sequence). A Petri net N is a tuple (S, T, F) with a set S of places, a set T of transitions, where $S \neq \emptyset \neq T$ and $S \cap T = \emptyset$, and a mapping $F: (S \times T) \cup (T \times S) \rightarrow \mathbb{N}$ defining arcs between places and transitions.

A marking or state of a Petri net is a map $m: S \rightarrow \mathbb{N}$. A place s is said to contain k tokens under m if $m(s) = k$. A transition $t \in T$ is enabled under m , $m[t\rangle$, if $m(s) \geq F(s, t)$ for every $s \in S$. A transition t fires under m and leads to m' , $m[t\rangle m'$, if additionally $m'(s) = m(s) - F(s, t) + F(t, s)$ for every $s \in S$.

A word $\sigma \in T^*$ is a firing sequence under m and leads to m' , $m[\sigma\rangle m'$, if either $m = m'$ and $\sigma = \varepsilon$, the empty word, or $\sigma = wt$, $w \in T^*$, $t \in T$ and $\exists m'': m[w\rangle m''[t\rangle m'$. A firing sequence σ under m is enabled under m , i.e. $m[\sigma\rangle$. The Parikh image of a word $\sigma \in T^*$ is the vector $\wp(\sigma): T \rightarrow \mathbb{N}$ with $\wp(\sigma)(t) = \#_t(\sigma)$, where $\#_t(\sigma)$ is the number of occurrences of t in σ . For any firing sequence σ , we call $\wp(\sigma)$ realizable.

As usual, places are drawn as circles (with tokens as black dots inside them), transitions as rectangles, and arcs as arrows with $F(x, y) > 0$ yielding an arrow pointing from x to y . If an arc has a weight of more than one, i.e. $F(x, y) > 1$, the number $F(x, y)$ is written next to the arc. In case $F(x, y) = F(y, x) > 0$, we may sometimes draw a line with arrowheads at both ends.

Note, that the Parikh image is not an injective function. Therefore, $\wp(\sigma)$ can be realizable even if σ is not a firing sequence (provided there is another firing sequence σ' with $\wp(\sigma) = \wp(\sigma')$).

Fig. 1. The word tt' cannot fire, but we can borrow a token from the circle uu' , so $utt'u'$ can fire and leads to the same marking as tt' . The incidence matrix of the net is shown on the right.

Definition 2 (Reachability problem). A marking m' is reachable from a marking m in a net $N = (S, T, F)$ if there is a firing sequence $\sigma \in T^*$ with $m[\sigma]m'$. A tuple (N, m, m') of a net and two markings is called a reachability problem and has the answer “yes” if and only if m' is reachable from m in N . The set $RP = \{(N, m, m') \mid N \text{ is a Petri net, } m' \text{ is reachable from } m \text{ in } N\}$ is generally called the reachability problem, for which membership is to be decided.

It is well-known that a necessary condition for a positive answer to the reachability problem is the feasibility of the *state equation*.

Definition 3 (State equation). For a Petri net $N = (S, T, F)$ let $C \in \mathbb{N}^{S \times T}$, defined by $C_{s,t} = F(t, s) - F(s, t)$, be the incidence matrix of N . For two markings m and m' , the system of linear equations $m + Cx = m'$ is the state equation of N for m and m' . A vector $x \in \mathbb{N}^T$ fulfilling the equation is called a solution.

Remark 1. For any firing sequence σ of a net $N = (S, T, F)$ leading from m to m' , i.e. $m[\sigma]m'$, holds $m + C\wp(\sigma) = m'$, i.e. the Parikh vector of σ is a solution of the state equation for N , m , and m' . This is just a reformulation of the firing condition for σ .

It is possible to have a sequence σ such that its Parikh image fulfills the state equation but it is not a firing sequence. The easiest example for this occurs in a net $N = (\{s\}, \{t\}, F)$ with $F(s, t) = 1 = F(t, s)$. Let m and m' be the *empty marking*, i.e. one with zero tokens overall, then $m[t]m'$ is obviously wrong but $m + C\wp(\sigma) = m'$ holds since $C = (0)$. The effect can occur whenever the Petri net contains a cycle of transitions. Interestingly, certain cycles of transitions can also help to overcome this problem, see Fig. 1. Here, we would like to fire a word tt' from the marking m with $m(s_1) = m(s_2) = 0$ and $m(s_3) = 1$, but obviously, this is impossible. If we borrow a token from s_3 , we can fire tt' , or more precisely $utt'u'$. As we return the borrowed token to s_3 in the end we reach the same marking tt' would have reached (if enabledness were neglected).

Definition 4 (T-invariant). Let $N = (S, T, F)$ be a Petri net and C its incidence matrix. A vector $x \in \mathbb{N}^T$ is called a *T-invariant* if $Cx = 0$.

A realizable *T-invariant* corresponds to a cycle in the state space. Its occurrence does not change the marking. However, its interleaving with another sequence σ may turn σ from unrealizable to realizable. The reason is that the partial occurrence of the

Fig. 2. Neither the T -invariant $\wp(tt')$ nor $\wp(uu')$ is realizable, but $\wp(tt'uu')$ is, by the sequence $tut'u'$

T -invariant may “lend” tokens to the otherwise blocked σ and can be completed after σ has produced another token on the same place later on.

Solving the state equation is a non-negative integer programming problem. From linear algebra we know that the solution space is semi-linear.

Corollary 1 (Solution space). *For a given state equation $m + Cx = m'$ over a net $N = (S, T, F)$, there are numbers $j, k \in \mathbb{N}$ and finite sets of vectors $B = \{b_i \in \mathbb{N}^T \mid 1 \leq i \leq j\}$ (base vectors) and $P = \{p_i \in \mathbb{N}^T \mid 1 \leq i \leq k\}$ (period vectors) such that:*

- all $b_i \in B$ are pairwise incomparable (by standard componentwise comparison for vectors) and thus minimal solutions,
- P forms a basis for the non-negative solution space $P^* = \{\sum_{i=1}^k n_i p_i \mid n_i \in \mathbb{N}, p_i \in P\}$ of $Cx = 0$,
- for all solutions x there are $n_i \in \mathbb{N}$ for $1 \leq i \leq k$ and $n \in \{1, \dots, j\}$ such that $x = b_n + \sum_{i=1}^k n_i p_i$,
- for every solution x , all vectors of the set $x + P^*$ are solutions as well.

Note that only linear combinations with nonnegative coefficients are considered in this representation.

So we know that all solutions can be obtained by taking a minimal solution b of the state equation and adding a linear combination of T -invariants from some basis P . Usually, not all the elements from B and P we use for a solution are realizable, though. While the sum of two realizable T -invariants remains realizable (just concatenate the according firing sequences as they have identical initial and final marking), the sum of two non-realizable T -invariants may well become realizable. This can be seen in Fig. 2, where neither $\wp(tt')$ nor $\wp(uu')$ is realizable under the marking m with $m(s_1) = m(s_4) = 1$ and $m(s_2) = m(s_3) = 0$, but the sequence $tut'u'$ realizes $\wp(tt'uu')$. The matter is even more complicated when a minimal solution from B is introduced, because positive minimal solutions are never T -invariants (unless $m = m'$), i.e. they change the marking of the net, so their realizations cannot just be concatenated.

3 Traversing the Solution Space

For solving the state equation an IP solver can be used. Fast IP solvers like *lp_solve* [1] allow to define an objective function – in our case to minimize the solution size and

obtain firing sequences that are as short as possible – and yield a single solution, at least if a solution exists. Fortunately, we can force an IP solver to produce more than just one solution — this is the CEGAR part of our approach. If a solution found is not realizable, we may add an inequation to our state equation to forbid that solution. Starting the IP solver again will then lead to a different solution. The trick is, of course, to add inequations in such a way that no realizable solution is lost.

Definition 5 (Constraints). Let $N = (S, T, F)$ be a Petri net. We define two forms of constraints, both being linear inequations over transitions:

- a jump constraint takes the form $t < n$ with $n \in \mathbb{N}$ and $t \in T$.
- an increment constraint takes the form $\sum_{i=1}^k n_i t_i \geq n$ with $n_i \in \mathbb{Z}$, $n \in \mathbb{N}$, and $t_i \in T$.

Jump constraints can be used to switch (jump) to another base solution, exploiting the incomparability of different minimal base solutions, while increment constraints are used to force non-minimal solutions. To understand the idea for differentiating between these two forms of constraints, it is necessary to introduce the concept of a partial solution first. A partial solution is obtained from a solution of the state equation under given constraints by firing as many transitions as possible.

Definition 6 (Partial solution). Let $N = (S, T, F)$ be a Petri net and Ω a total order over $\mathbb{N}^T$ that includes the partial order given by $x < y$ if $\sum_{t \in T} x(t) < \sum_{t \in T} y(t)$. A partial solution of a reachability problem (N, m, m') is a tuple $(\mathcal{C}, x, \sigma, r)$ of

- a family of (jump and increment) constraints $\mathcal{C} = (c_1, \dots, c_n)$,
- the Ω -smallest solution x fulfilling the state equation of (N, m, m') and the constraints of $\mathcal{C}$,
- a firing sequence $\sigma \in T^*$ with $m[\sigma]$ and $\wp(\sigma) \leq x$,
- a remainder r with $r = x - \wp(\sigma)$ and $\forall t \in T: (r(t) > 0 \implies \neg m[\sigma t])$.

The vectors x and r are included for convenience only, they can be computed from $\mathcal{C}$, σ , Ω , and the problem instance.

A full solution is a partial solution $(\mathcal{C}, x, \sigma, r)$ with $r = 0$. In this case, σ is a firing sequence solving our reachability problem (with answer 'yes').

We choose Ω such that an IP solver can be assumed to always produce the Ω -smallest solution that does not contradict its linear system of equations.

Corollary 2 (Realizable solutions are full solutions). For any realizable solution x of the state equation we find a full solution $(\mathcal{C}, x, \sigma, \emptyset)$ where $\mathcal{C}$ consists of constraints $t \geq x(t)$ for every t with $x(t) > 0$, and $\wp(\sigma) = x$.

Note, that x is the smallest solution fulfilling c and therefore also the Ω -smallest solution.

By adding a constraint to a partial solution we may obtain new partial solutions (or not, if the linear system becomes infeasible). Any full solution can eventually be reached by consecutively extending an Ω -minimal partial solution with constraints.

Fig. 3. Paths from the Ω -minimal solution b to any solution. Black dots represent solutions, cones stand for linear solution spaces over such solutions, which may or may not intersect or include each other. Normal arrows increment a solution by adding a T -invariant, dashed arrows are jumps to an incomparable Ω -greater solution. Such jumps can also occur on higher levels of linear solution spaces, shown by the dotted arrow.

Lemma 1 (A path to a full solution). *Let b be the Ω -minimal solution of the state equation of a reachability problem (N, m, m') and $ps' = ((c_j)_{1 \leq j \leq \ell}, b' + \sum_{i=1}^k n_i p_i, \sigma', 0)$ a full solution of the problem. For $0 \leq n \leq \ell$, there are partial solutions $ps_n = ((c_j)_{1 \leq j \leq n}, x_n, \sigma_n, r_n)$ with $ps_0 = (\emptyset, b, \sigma_0, r_0)$, $ps_\ell = ps'$, and $x_{n_1} \leq_\Omega x_{n_2}$ for $n_1 \leq n_2$.*

Proof. Let $\mathcal{C}_n = (c_j)_{1 \leq j \leq n}$. If ps_{n_1}, ps_{n_2} are two partial solutions (with $1 \leq n_1 < n_2 \leq \ell$) then x_{n_2} is a solution of the state equation plus $\mathcal{C}_{n_1}$, since it even fulfills the state equation plus $\mathcal{C}_{n_2}$ with $\mathcal{C}_{n_1} \subseteq \mathcal{C}_{n_2}$. As x_{n_1} is the Ω -smallest solution of the state equation plus $\mathcal{C}_{n_1}$, $x_{n_1} \leq_\Omega x_{n_2}$ holds. Therefore, $b \leq_\Omega x_1 \leq_\Omega \dots \leq_\Omega x_\ell$. Since $x_\ell = b' + \sum_{i=1}^k n_i p_i$ is an existing solution of the strictest system, i.e. state equation plus $\mathcal{C}_\ell$, each system of state equation plus one family of constraints $\mathcal{C}_n$ is solvable. As a σ_n can be determined by just firing transitions as long as possible, all the partial solutions ps_n exist.

Now, let us assume a partial solution $ps = (\mathcal{C}, x, \sigma, r)$ that is not a full solution, i.e. $r \neq 0$. Obviously, some transitions cannot fire often enough. There are three possible remedies for this situation:

1. If x is realizable, we can find a full solution $ps' = (\mathcal{C}, x, \sigma', 0)$ with $\wp(\sigma') = x$.
2. We can add a jump constraint to obtain an Ω -greater solution vector for a different partial solution.
3. If $r(t) > 0$ for some transition t , we can add an increment constraint to increase the maximal number of tokens available on a place in the preset of t . Since the final marking remains the same, this means to borrow tokens for such a place. This can be done by adding a T -invariant containing the place to the solution.

A visualization of these ideas can be seen in Fig. 3 where b denotes the Ω -smallest solution. The cone over b represents all solutions $b + P^*$ with P being the set of period vectors, i.e. T -invariants. Jump constraints lead along the dashed or dotted lines to the next Ω -minimal solution while normal arrows representing increment constraints lead upwards to show the addition of a T -invariant. How to build constraints doing just what we want them to do is the content of the next section.

4 Building Constraints

Let us first argue that for a state equation, any of the minimal solution vectors in B can be obtained by using jump constraints.

Lemma 2 (Jumps to minimal solutions). *Let $b, b' \in B$ be base vectors of the solution space of the state equation $m + Cx = m'$ plus some set of constraints $\mathcal{C}$. Assume b to be the Ω -minimal solution of the system. Then, we can obtain b' as output of our IP solver by consecutively adding jump constraints of the form $t_i < n_i$ with $n_i \in \mathbb{N}$ to $\mathcal{C}$.*

Proof. We know $b \leq_{\Omega} b'$ holds, but since b' is a minimal solution, $b \leq b'$ cannot hold. Therefore, a transition t with $b'(t) < b(t)$ must exist. After adding the constraint $t < b(t)$ to $\mathcal{C}$ the IP solver can no longer generate b as a solution. Assume b'' is the newly generated solution. If $b' = b''$ we are done. Otherwise, since b' fulfills $t < b(t)$, it is still a solution of our system, and also a minimal one as the solution space is restricted by the added constraint. Thus, $b'' \leq_{\Omega} b'$ holds and we may recursively use the same argument as above for $b := b''$. Since there are only finitely many solutions Ω -smaller than b' , the argument must terminate reaching b' .

Non-minimal solutions may not be reachable this way, since the argument “ $b'(t) < b(t)$ for some t ” does not necessarily hold. We will need increment constraints for this, but unluckily, increment constraints and jump constraints may contradict each other. Assume our state equation has a solution of the form $b' + p$ with a period vector $p \in P$ and to obtain $b' \in B$ from the Ω -minimal solution $b \in B$ we need to add (at least) a jump constraint $t_i < n_i$ to the state equation. If p contains t_i often enough, we will find that $(b' + p)(t_i) \geq n_i$ holds. Therefore, $b' + p$ is not a solution of the state equation plus the constraint $t_i < n_i$, i.e. adding an increment constraint demanding enough occurrences of t_i for $b' + p$ will render the linear equation system infeasible. The only way to avoid this problem is to remove the jump constraints before adding increment constraints.

Lemma 3 (Transforming jumps). *Let z be the Ω -minimal solution of the state equation $m + Cx = m'$ plus some constraints $\mathcal{C}$. Let $\mathcal{C}'$ consist of all increment constraints of $\mathcal{C}$ plus a constraint $t \geq z(t)$ for each transition t . Then, for all $y \geq z$, y is a solution of $m + Cx = m'$ plus $\mathcal{C} \cap \mathcal{C}'$ if and only if y is a solution of $m + Cx = m'$ plus $\mathcal{C}'$. Furthermore, no Ω -smaller solution of $m + Cx = m'$ plus $\mathcal{C}$ than z solves $m + Cx = m'$ plus $\mathcal{C}'$.*

Proof. Let $y \geq z$ be a solution of $m + Cx = m'$ plus $\mathcal{C} \cap \mathcal{C}'$. The additional constraints in $\mathcal{C}'$ only demand $y(t) \geq z(t)$, which is obviously the case. The other direction is trivial. For the second part, let $z' \leq_{\Omega} z$ with $z \neq z'$ be some solution of $m + Cx = m'$ plus $\mathcal{C}$. Since $\sum_t z'(t) \leq \sum_t z(t)$ (following from Ω) but $z \neq z'$, for at least one transition t holds $z'(t) < z(t)$. Due to the constraint $t \geq z(t)$ in $\mathcal{C}'$, z' cannot be a solution of $m + Cx = m'$ plus $\mathcal{C}'$.

As a consequence, if we are only interested in solutions of the cone $z + P^*$ over z , we can add increment constraints guaranteeing solutions greater or equal than z and remove all jump constraints without any further restriction. Our IP solver will yield z

as the Ω -minimal solution for both families of constraints, $\mathcal{C}$ and $\mathcal{C}'$, and we can add further constraints leading us to any solution in the cone $z + P^*$ now.

Let $ps = (\mathcal{C}, x, \sigma, r)$ now be a partial solution with $r > 0$. We would like to determine sets of places that need additional tokens (and the number of these tokens) that would enable us to fire the remainder r of transitions. Obviously, this problem is harder than the original problem of finding out if a transition vector is realizable, i.e. just testing if zero additional tokens are sufficient. A recursive approach would probably be very inefficient as for every solution x there may be many different remainders r . Even though the remainders are smaller than the solution vector x , the number of recursion steps might easily grow exponentially with the size of x , i.e. $\sum_t x(t)$. We therefore adopt a different strategy, namely finding good heuristics to estimate the number of tokens needed. If a set of places actually needs n additional tokens with $n > 0$, our estimate may be any number from one to n . If we guess too low, we will obtain a new partial solution allowing us to make a guess once again, (more or less) slowly approaching the correct number. We propose a two-part algorithm, the first part dealing with sets of places and transitions that are of interest.

input: Reachability prob. (N, m, m') ; partial solution $ps = (\mathcal{C}, x, \sigma, r)$
output: A set of tuples (S_i, T_i, X_i) with $S_i \subseteq S$, $T_i \cup X_i \subseteq T$
Determine $\hat{m}$ with $m[\sigma]\hat{m}$;
Build a bipartite graph $G = (S_0 \cup T_0, E)$ with
 $T_0 := \{t \in T \mid r(t) > 0\}$; $S_0 := \{s \in S \mid \exists t \in T_0 : F(s, t) > \hat{m}(s)\}$;
 $E := \{(s, t) \in S_0 \times T_0 \mid F(s, t) > \hat{m}(s)\} \cup \{(t, s) \in T_0 \times S_0 \mid F(t, s) > F(s, t)\}$;
Calculate the strongly connected components (*SCCs*) of G ;
 $i := 1$;
for each source *SCC* (i.e. one without incoming edges):
 $S_i := SCC \cap S_0$;
 $T_i := SCC \cap T_0$;
 $X_i := \{t \in T_0 \setminus SCC \mid \exists s \in S_i : (s, t) \in E\}$;
 $i := i + 1$;
end for

The edges of the graph G constructed in the algorithm have a different meaning depending on their direction. Edges from transitions to places signal that the transition would increase the number of tokens on the place upon firing, while edges in the other direction show the reason for the non-enabledness of the transition. A source SCC, i.e. a strongly connected component without incoming edges from other components, can therefore not obtain tokens by the firing of transitions from other SCCs. This means, tokens must come from somewhere else, that is, from firing transitions not appearing in the remainder r . For each set of places S_i such identified as non-markable by the remainder itself, there are two sets of transitions. If one transition from the set T_i would become fireable, it is possible that all other transitions could fire as well, since the former transition effectively produces tokens on some place in the component. If the set T_i is empty (the SCC consisting of a single place), the token needs of all the transitions in X_i together must be fulfilled, since they cannot activate each other. We can thus calculate how many tokens we need at least:

input: A tuple (S_i, T_i, X_i) ; (N, m, m') and $\hat{m}$ from above
output: A number of tokens n (additionally needed for S_i)
if $T_i \neq \emptyset$
then $n := \min_{t \in T_i} (\sum_{s \in S_i} (F(s, t) - \hat{m}(s)))$
else sort X_i in groups $G_j := \{t \in X_i \mid F(t, s) = j\}$ (with $S_i = \{s\}$);
 $n := 0$; $c := 0$;
for j **with** $G_j \neq \emptyset$ **downwards loop**
 $c := c - j * (|G_j| - 1) + \sum_{t \in G_j} F(s, t)$;
if $c > 0$ **then** $n := n + c$ **end if**;
 $c := -j$
end for
end if

Note that the transitions in X_i all effectively consume tokens from $s \in S_i$, but they may leave tokens on this place due to a loop. By firing those transitions with the lowest $F(t, s)$ -values last, we minimize the leftover. Transitions with the same $F(t, s)$ -value j can be processed together, each consuming effectively $F(s, t) - j$ tokens except for the “first” transition which will need j more tokens. If some group G_j of transitions leaves tokens on s , the next group can consume them, which is memorized in the variable c (for carryover or consumption). Observe, that the algorithm cannot return zero: There must be at least one transition in $T_i \cup X_i$, otherwise there would be no transition that cannot fire due to a place in S_i and the places in S_i would not have been introduced at all. If T_i is not empty, line 4 in the algorithm will minimize over positive values; if T_i is empty, line 8 will set c to a positive value at its first execution, yielding a positive value for n . Overall, our argumentation shows:

Corollary 3. *For each set of places S_i that need additional tokens according to the first part of the algorithm, the second part estimates that number of tokens to be in a range from one to the actual number of tokens necessary.*

We can thus try to construct a constraint from a set of places S_i generated by the first part of the algorithm and the token number calculated in the second part. Since our state equation has transitions as variables, we must transform our condition on places into one on transitions first.

Corollary 4. *Let $N = (S, T, F)$ be a Petri net, (N, m, m') the reachability problem to be solved, $ps = (\mathcal{C}, x, \sigma, r)$ a partial solution with $r > 0$, and $\hat{m}$ the marking reached by $m[\sigma]\hat{m}$. Let S_i be a set of places and n a number of tokens to be generated on S_i . Further, let $T_i := \{t \in T \mid r(t) = 0 \wedge \sum_{s \in S_i} (F(t, s) - F(s, t)) > 0\}$. We define a constraint c by*

$$\sum_{t \in T_i} \sum_{s \in S_i} (F(t, s) - F(s, t))t \geq n + \sum_{t \in T_i} \sum_{s \in S_i} (F(t, s) - F(s, t))\wp(\sigma)(t).$$

Then, for the system $m + Cx = m'$ plus $\mathcal{C}$ plus c , if our IP solver can generate a solution $x + y$ (y being a T -invariant) we can obtain a partial solution $ps' = (\mathcal{C} \cup \{c\}, x + y, \sigma\tau, r + z)$ with $\wp(\tau) + z = y$. Furthermore, $\sum_{t \in T} \sum_{s \in S_i} (F(t, s) - F(s, t))y(t) \geq n$.

First, note that T_i contains the transitions that produce more on S_i than they consume, but we have explicitly excluded all transitions of the remainder r , since we do not want the IP solver to increase the token production on S_i by adding transitions that could not fire anyway. I.e., we would like to have a chance to fire the additional transitions in y at some point, though there are no guarantees. The left hand side of c contains one instance of a transition t for each token that t effectively adds to S_i . If we apply some transition vector x to the left hand side of c , we therefore get the number of tokens added to S_i by the transitions from T_i in x . Of course, other transitions in x might reduce this number again. For the right hand side of c , we calculate how many tokens are actually added to S_i by the transitions from T_i in the firing sequence σ (and therefore also in the solution x) and increase that number by the n extra tokens we would like to have. Since the extra tokens cannot come from x in a solution $x + y$, they must be produced by y , i.e. $\sum_{t \in T} \sum_{s \in S_i} (F(t, s) - F(s, t))y(t) \geq n$. We might be able to fire some portion of y after σ , resulting in the obvious $\wp(\tau) + z = y$. When we apply our constraint we might get less or more than the n extra tokens, depending on the T -invariants in the net. Further constraints may or may not help. At this point we can state:

Theorem 1 (Reachability of solutions). *Every realizable solution of the solution space of a state equation can be reached by consecutively adding constraints to the system of equations, always transforming jump constraints before adding increment constraints.*

5 Finding Partial Solutions

Producing partial solutions $ps = (\mathcal{C}, x, \sigma, r)$ from a solution x of the state equation (plus $\mathcal{C}$) is actually quite easily done by brute force. We can build a tree with marking-annotated nodes and the firing of transitions as edges, allowing at most $x(t)$ instances of a transition t on any path from the root of the tree to a leaf. Any leaf is a new partial solution from which we may generate new solutions by adding constraints to the state equation and forwarding the evolving linear system to our IP solver. If we just make a depth-first-search through our tree and backtrack at any leaf, we build up all possible firing sequences realizable from x . This is obviously possible without explicitly building the whole tree at once, thus saving memory. Of course, the tree might grow exponentially in the size of the solution vector x and so some optimizations are in order to reduce the run-time. We would like to suggest a few ones here, especially partial order reductions.

1. The stubborn set method [8] determines a set of transitions that can be fired before all others by investigating conflicts and dependencies between transitions at the active marking. The stubborn set is often much smaller than the set of enabled transitions under the same marking, leading to a tree with a lower degree. In our case, in particular the version of [13] is useful as, using this method, the reduced state space contains, for each trace to the target marking, at least one permutation of the same trace. Hence, the reduction is consistent with the given solution of the state equation.
2. Especially if transitions should fire multiple times ($x(t) > 1$) we observe that the stubborn set method alone is not efficient. The situation in Fig. 4 may occur quite

Fig. 4. If both sequences $\alpha t \sigma u$ and $\alpha u \sigma t$ can be fired, the subtrees after the nodes with marking $\tilde{m}$ are identical. Only one of the subtrees needs to be evaluated, the other one may be omitted. Snaked lines denote firing sequences.

often. Assume we reach some marking $\hat{m}$ by a firing sequence α , so that transitions t and u are enabled. After proceeding through the subtree behind t we backtrack to the same point and now fire u followed by some sequence σ after which t is enabled, leading to $m[\alpha]\hat{m}[u\sigma t]\tilde{m}$. If $\hat{m}[t\sigma u]$ holds, we know that it reaches the same marking $\tilde{m}$ and the same remainder r of transitions still has to fire. Therefore, in both cases the future is identical. Since we have already investigated what happens after firing $\alpha t \sigma u$, we may backtrack now omitting the subtree after $\alpha u \sigma t$. Note that a test if $\hat{m}[t\sigma u]$ holds is quite cheap, as only those places s with $C_{s,t} < C_{s,u}$ can prevent the sequence $t\sigma$. Enabledness of u after $t\sigma$ can be tested by reverse calculating $\tilde{m}_1 = \tilde{m} - Cu$ and checking whether $\tilde{m}_1$ is a marking and $\tilde{m}_1[u]\tilde{m}$ holds.

3. There are situations where a leaf belongs to a partial solution ps' that cannot lead to a (new) full solution. In this case the partial solution does not need to be processed. If we already tried to realize x yielding a partial solution $ps = (\mathcal{C}, x, \sigma, r)$ and $ps' = (\mathcal{C} \cup \{c\}, x + y, \sigma, r + y)$ is our new partial solution with an increment constraint c and a T -invariant y , any realizable solution $x + y + z$ obtainable from ps' can also be reached from ps by first adding a constraint c' for the T -invariant z (and later c, y). If no transition of z can be fired after σ , $y + z$ is also not realizable after firing σ . We may be able to mingle the realization of z with the firing of σ , but that will be reflected by alternate partial solutions (compared to both, ps and ps'). Therefore, not processing ps' will not lose any full solutions.
4. A similar situation occurs for $ps' = (\mathcal{C} \cup \{c\}, x + y, \sigma\tau, r)$ with $\wp(\tau) = y$. There is one problem, though. Since we estimated a token need when choosing c and that estimate may be too low, it is possible that while firing τ we get closer to enabling some transition t in r without actually reaching that limit where t becomes firable. We thus have to check for such a situation (by counting the minimal number of missing tokens for firing t in the intermediate markings occurring when firing σ and τ). If τ does not help in approaching enabledness of some t in r , we do not need to process ps' any further.
5. Partial solutions should be memorized if possible to avoid using them as input for CEGAR again if they show up more than once.

6 Experimental Results

The algorithm presented here has been implemented in a tool named Sara [16]. We compare Sara to LoLA [17], a low level analyzer searching the (reduced) state space

of a Petri net. According to independent reports, e.g. [15], LoLA performs very well on reachability queries and possibly is the fastest tool for standard low level Petri nets. The following tests, real-world examples as well as academic constructions, were run on a 2.6GHz PC with 4GB RAM under Windows XP and Cygwin. While the CPU had four cores, only one was used for the tools. Tests on a similar Linux system lead to comparable but slightly faster results.

- 590 business processes with about 20 up to 300 actions each were tested for “relaxed soundness”. The processes were transformed into Petri nets and for each action a test was performed to decide if it was possible to execute the action and reach the final state of the process afterwards. Successful tests for all actions/transitions yield relaxed soundness. Sara was able to decide relaxed soundness for all of the 590 nets together (510 were relaxed sound) in 198 seconds, which makes about a third of a second per net. One business process was especially hard and took 12278 calls to `lp_solve` and 24 seconds before a decision could be made. LoLA was unable to solve 17 of the problems (including the one mentioned above) and took 24 minutes for the remaining 573.
- Four Petri nets derived in the context of verifying parameterized boolean programs (and published on a web page [6]) were presented to us to decide coverability. Sara needed less than one time slice of the CPU per net and solved all instances correctly. LoLA was not able to find the negative solution to one of the problems due to insufficient memory (here, tests were made with up to 32GB RAM), the remaining three problems were immediately solved.
- In 2003, H. Garavel [5] proposed a challenge on the internet to check a Petri net derived from a LOTOS specification for dead (i.e. never firable) transitions. The net consisted of 776 transitions and 485 places, so 776 tests needed to be made. Of the few tools that succeeded, LoLA was the fastest with about 10 minutes, but it was necessary to handle two of the transitions separately with a differently configured version of LoLA. In our setting, seven years later, LoLA needed 41 seconds to obtain the same result. Sara came to the same conclusions in 26 seconds. In most cases the first solution of `lp_solve` was sufficient, but for some transitions it could take up to 15 calls to `lp_solve`. Since none of the 776 transitions is dead, Sara also delivered 776 firing sequences to enable the transitions, with an average length of 15 and a longest sequence of 28 transitions. In 2003 the best upper bound for the sequences lengths was assumed to be 35, while LoLA found sequences of widely varying length, though most were shorter than 50 transitions.
- Using specifically constructed nets with increasing arc weights (and token numbers) it was possible to outsmart Sara – the execution times rose exponentially with linearly increasing arc weights, the first five times being 0.1, 3.3, 32, 180, and 699 seconds. LoLA, on the other hand, decided reachability in less than 3 seconds (seemingly constant time) in these cases.

We also checked our heuristics from Sec. 5 with some of the above nets by switching the former off and comparing the results (see Table 1). Our implementation needs both forms of constraints, jump and increment, to guarantee that all solutions of the state equation can be visited. Going through these solutions in a different order, e.g. the total order Ω , is difficult and a comparison was not possible so far.

Table 1. Results for shutting down one heuristic. Inst. is the number of problem instances to be solved for the net, Sol? the average solution length or “-” if no solution exists. Columns Full, $\neg 1$, $\neg 2$, $\neg 3/4$, and $\neg 5$ contain the result with all optimizations, without stubborn sets, without subtree cutting, without partial solution cutting, and without saving intermediate results (numbers are according to Sec. 5). Each entry shows the elapsed time and the number of necessary CEGAR steps (average), or NR if no result could be obtained in less than a day.

Net	Inst.	Sol?	Full	$\neg 1$	$\neg 2$	$\neg 3/4$	$\neg 5$
garavel	776	15	26s (0.11)	25s (0.11)	26s (0.11)	26s (0.11)	26s (0.11)
bad-bp	142	-	24s (85)	24s (85)	24s (85)	24s (85)	NR
good-bp	144	53	1.7s (0)	1.7s (0)	1.7s (0)	1.7s (0)	1.7s (0)
test7	10	175	29s (13)	990s (22)	NR	49s (14)	29s (13)
test8-1	1	40	0.1s (13)	0.35s (22)	49s (13)	0.2s (14)	0.11s (13)
test8-2	1	76	3.3s (21)	24s (51)	NR	11s (34)	3.8s (21)
test8-3	1	112	32s (27)	390s (80)	NR	175s (71)	33s (27)
test9	1	-	0.4s (53)	22s (464)	NR	NR	0.9s (65)

The nets tested fall in two categories. Garavel’s net and the business processes are extensive nets with a low token count and without much concurrency that could be tackled by partial order reduction. The heuristics have no effect here, short runtimes result from finding a good solution to the state equation early on. Only for the hardest of the business processes (bad-bp) memorizing intermediate results to avoid checking the same partial solution over and over made sense – without it we did not get a result at all.

The other category are compact nets. In our test examples a high number of tokens is produced and then must be correctly distributed, before the tokens can be removed again to produce the final marking. With a high level of concurrency in the nets, partial order reduction is extremely useful, the cutting off of already seen subtrees(2) even more than the stubborn set method(1). In the last net (test9), the sought intermediate token distribution is unreachable but the state equation has infinitely many solutions. Only by cutting off infinite parts of the solution tree with the help of optimization 3 and 4 it becomes possible to solve the problem at all. Without them, the number of outstanding CEGAR steps reaches 1000 within less than a minute and continues to increase monotonically. The algorithm slows down more and more then as the solutions to the state equation and thus the potential firing sequences become larger.

Beyond what other tools can do, namely solving the problem and – in the positive case – present a witness path, i.e. firing sequence, Sara can also provide diagnostic information in the negative case as long as the state equation has a solution. This feature was tested e.g. with the hardest of the 590 business processes from above, which provides such a negative case for some of its 142 transitions. Since we cannot present such a large net here, a condensed version with the same important features is shown in Fig. 5.

Sara provides a partitioning of the net showing where the relaxed soundness test (for any of the transitions k_1 , k_2 , or x_2) fails, e.g. it is impossible to fire x_2 and afterwards reach the final marking with exactly one token on place o (other places being empty). The solution $d + k_1 + k_2 + x_2$ of the state equation can neither be realized nor extended to a “better” solution. The ascending pattern shows a region of the net (given by Sara)

Fig. 5. A condensed, flawed business process. One token should flow from the initial place i to the output place o with all other places empty finally. Non-white transitions appear in Sara's solution to the state equation, but only the dark gray one is firable. Ascending stripes show the area with non-firable transitions where additional tokens could not be generated.

where tokens are needed but cannot be generated without violating the state equation. The descending pattern marks areas that are affected by the former ones, i.e. areas with also non-firable transitions. The gray transition d is the only firable transition occurring in the solution. When analyzing the net we can see that the cycle $c_1 - k_1 - c_2 - k_2$ indeed constitutes a flaw for a business process: if the cycle gets marked and then emptied later, at least two tokens must flow through a_2 , one of which can never be removed. Using u instead of d is therefore impossible, i.e. dx_1 is the only firing sequence reaching the final marking.

7 Conclusion

We proposed a promising technique for reachability verification. For reachable problem instances, it tends to yield short witness paths. For unreachable instances, it may terminate early, without an exhaustive search. Furthermore, it may provide some diagnostic information in that case. Our approach applies the concept of counterexample guided abstraction refinement in a novel context: the abstraction is not given as a transition system but as a linear-algebraic overapproximation of the reachable states. In essence, we replace the search in the set of states by the more focussed search through the solutions of the state equation.

The state equation as such has been used earlier for verification purposes, see for instance [4]. In [14], it is used as an initial way of narrowing the state space exploration but not refined according to the CEGAR.

References

1. Berkelaar, M., Eikland, K., Notebaert, P.: *lp_solve Reference Guide* (2010), <http://lpsolve.sourceforge.net/>
2. Ciardo, G., Marmorstein, R., Siminiceanu, R.: The saturation algorithm for symbolic state space exploration. *Software Tools for Technology Transfer* 8(1), 4–25 (2006)

3. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855. Springer, Heidelberg (2000)
4. Esparza, J., Melzer, S., Sifakis, J.: Verification of safety properties using integer programming: Beyond the state equation. *Formal Methods in System Design* 16 (2), 159–189 (2000)
5. Garavel, H.: Efficient Petri Net tool for computing quasi-liveness (2003), <http://www.informatik.uni-hamburg.de/cgi-bin/TGI/pnml/getpost?id=2003/07/2709>
6. Geeraerts, G., Raskin, J.F., Van Begin, L.: Expand, enlarge and check (2010), <http://www.ulb.ac.be/di/ssd/ggeeraer/eec/>
7. Kosaraju, S.R.: Decidability of reachability in vector addition systems. In: *Proceedings of the 14th Annual ACM STOC*, pp. 267–281 (1982)
8. Kristensen, L.M., Schmidt, K., Valmari, A.: Question-guided Stubborn Set Methods for State Properties. *Formal Methods in System Design* 29(3), 215–251 (2006)
9. Lambert, J.L.: A structure to decide reachability in Petri nets. *Theoretical Computer Science* 99, 79–104 (1992)
10. Leroux, J.: The General Vector Addition System Reachability Problem by Presburger Inductive Invariants. In: *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science*, pp. 4–13. IEEE Computer Society, Los Alamitos (2009)
11. Lipton, R.J.: The Reachability Problem Requires Exponential Space. *Research Report* 62 (1976)
12. Mayr, E.: An algorithm for the general Petri net reachability problem. *SIAM Journal of Computing* 13(3), 441–460 (1984)
13. Schmidt, K.: Stubborn sets for standard properties. In: Donatelli, S., Kleijn, J. (eds.) ICATPN 1999. LNCS, vol. 1639, pp. 46–65. Springer, Heidelberg (1999)
14. Schmidt, K.: Narrowing petri net state spaces using the state equation. *Fundamenta Informaticae* 47(3–4), 325–335 (2001)
15. Talcott, C., Dill, D.: The pathway logic assistant. In: *Third International Workshop on Computational Methods in Systems Biology* (2005)
16. Wimmel, H.: Sara – Structures for Automated Reachability Analysis (2010), <http://service-technology.org/tools/download>
17. Wolf, K.: Generating Petri net state spaces. In: Kleijn, J., Yakovlev, A. (eds.) ICATPN 2007. LNCS, vol. 4546, pp. 29–42. Springer, Heidelberg (2007)