



Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Extensions to the CEGAR Approach on Petri Nets

BACHELOR'S THESIS

Author

Ákos HAJDU

Supervisors

dr. Tamás BARTHA, András VÖRÖS

December 12, 2013

Contents

Kivonat	5
Abstract	6
1 Introduction	7
2 Background	9
2.1 Petri nets	9
2.1.1 Definition	9
2.1.2 Incidence matrix and state equation	11
2.1.3 Inhibitor arcs	13
2.2 Reachability analysis	14
2.2.1 Reachability problem	14
2.2.2 Submarking coverability problem	14
2.2.3 Necessary and sufficient criteria	15
2.2.4 Decidability and complexity	15
2.2.5 Existing methods for solving reachability	16
2.3 Linear programming (LP)	16
2.3.1 Integer Linear Programming (ILP)	17
3 CEGAR approach on Petri nets	18
3.1 The CEGAR approach	18
3.1.1 Abstraction	18
3.1.2 Counterexample guided refinement	19
3.2 Reachability analysis of Petri nets using CEGAR	19
3.2.1 Solution space of the state equation	20
3.2.2 Partial solutions	21
3.2.3 Generating constraints	23
3.2.4 Reachability of solutions	26
3.2.5 Optimizations	26
3.2.6 A complex example	28
4 Algorithmic contributions	31
4.1 Extensions of the algorithm	31

4.1.1	Solving submarking coverability	31
4.1.2	Handling inhibitor arcs	31
4.2	Correctness of the algorithm	32
4.2.1	Proof of the incorrectness	32
4.2.2	Providing a solution in case of over-estimation	34
4.2.3	Detecting over-estimation structurally	35
4.3	Completeness of the algorithm	37
4.3.1	Previous work	37
4.3.2	Involving distant T-invariants	39
4.3.3	Extending the filtering optimization	43
4.4	Pseudo code	44
5	Implementation	48
5.1	The PetriDotNet framework	48
5.2	Architecture	49
5.3	Functionality	49
5.3.1	Deployment	50
5.3.2	Overview of the GUI	50
5.3.3	Information about the net	51
5.3.4	Parameters of the reachability problem	51
5.3.5	Configuration of the algorithm	53
5.3.6	Examination of the result of the algorithm	53
5.4	Development	55
5.4.1	lpsolve	55
5.4.2	Classes	56
6	Evaluation	60
6.1	Scalability	60
6.1.1	Counter	60
6.1.2	Dining philosophers	60
6.1.3	FMS	62
6.1.4	Kanban	63
6.1.5	Slotted ring	63
6.2	Comparison to the saturation algorithm	64
7	Conclusions	65
	Acknowledgment	67
	List of figures	68
	List of tables	70
	Bibliography	71

HALLGATÓI NYILATKOZAT

Alulírott *Hajdu Ákos*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, December 12, 2013

Hajdu Ákos
hallgató

Kivonat

Napjainkban a formális verifikáció a hibamentesség igazolásának egyre gyakrabban alkalmazott módszere, különösen a biztonságkritikus rendszerek területén, ahol matematikailag precíz módon kell bizonyítani a specifikációnak való megfelelést és a hibátlan működést. A formális módszerek nagy hátránya azonban a nem triviális méretű problémáknál fellépő, gyakran hatalmas számítási igényük. Ez a probléma az egyik legfontosabb formális verifikációs módszernél, az elérhetőségi analízisnél is előjön: egészen kicsi modelleknek is lehet óriási, vagy akár végtelen nagy állapottere. Ezen probléma leküzdésére mindig újabb és hatékonyabb algoritmusokra van szükség, hiszen a módszerek fejlődésével párhuzamosan a megoldandó problémák komplexitása is egyre nő. Az egyik ilyen algoritmus az úgynevezett ellenpélda-alapú absztrakció finomítás (CEGAR) módszert alkalmazza. Lényege, hogy az eredeti modell egy absztrakcióján dolgozik, és igény esetén finomítja az absztrakciót az állapottér bejárása során szerzett információk alapján. Az absztrakt modell állapotterének reprezentációja általában egyszerűbb, mint az eredeti modellel volt, így az absztrakt probléma könnyebben megoldható.

Szakedolgozatomban egy, a közelmúltban publikált algoritmust vizsgálok, amely az ellenpélda-alapú absztrakció finomítást a Petri-hálók elérhetőségi analízisére alkalmazza. Megmutatom, hogy bizonyos speciális esetekben az algoritmus helytelen eredményt adhat, és számos ellenpéldát mutatok be az algoritmus teljességére. Szakedolgozatomban új fejlesztéseket dolgozok ki, amelyek biztosítják az algoritmus helyességét és szélesítik az eldönthető problémák körét. Az algoritmust és az új fejlesztéseket implementáltam, dolgozatomban mérésekkel vizsgálom meg a kidolgozott új eljárások teljesítményét.

Abstract

Formal verification is becoming more prevalent, especially in the development of safety-critical systems where mathematically precise proofs are required to ensure suitability and faultlessness. The major drawback of formal methods is their computation intensive nature. This also holds for one of the most important formal verification technique, the reachability analysis: even a relatively small model can have a large or infinite state space. In order to overcome this problem, new and efficient algorithms are required, since the complexity of the models also tend to increase. One of these algorithms is the so called counterexample guided abstraction refinement (CEGAR) method. The CEGAR approach works on an abstraction of the original model and refines the abstraction using the information from the explored part of the state space. The abstract model usually has a less detailed state space representation. Therefore, the reachability analysis on the abstract model is easier.

In my work I examine a recently published algorithm, which applies the CEGAR approach on the reachability analysis of Petri nets. I show that the algorithm can give an incorrect answer in some special cases and I also present counterexamples of the completeness of the algorithm. I suggest new improvements to ensure correctness and to extend the set of decidable problems. I also implemented the algorithm with the new contributions and I analyze its performance by measurements.

Chapter 1

Introduction

Development of complex, distributed and safety-critical systems require mathematically precise verification techniques in order to ensure suitability and faultlessness. Formal modeling and analysis provide such tools. However, a major drawback of formal methods is their high computational complexity. Even for relatively small models, the set of possible states and behaviors can be unmanageably large or even infinite.

This also holds for one of the most popular modeling formalisms, Petri nets. Petri nets are widely used to model asynchronous, distributed, parallel and non-deterministic systems. The behavior of a Petri net model is determined by the set of possible states and transitions, i.e., the state space. One of the most important formal verification technique involving Petri nets is the *reachability analysis*, which checks if a given state is reachable from the initial state of the model. The reachability problem belongs to the mathematically hard problems. It is decidable, but no upper bound of its complexity is known and the lower bound is at least EXPSPACE-hard.

Counterexample guided abstraction refinement (CEGAR) is a general approach for solving hard problems. It works on an abstraction of the original model, which has a less detailed representation. Therefore, the abstract problem can be solved easier. However, due to the abstraction, a behavior in the abstract model may not be realizable in the original one. In this case the abstraction has to be refined using the information from the explored part of the state space. Recently, a new algorithm was published [1], which applies the CEGAR approach on the reachability problem of Petri nets.

In our previous work [2, 3] we examined the algorithm and proved that it is incorrect in certain situations, and we provided a method to detect such situations. We also proved the incompleteness of the algorithm by several examples, and suggested solutions to most of them. However, there are some problems that even the improved algorithm cannot solve.

In my current work I did further examination of the correctness and completeness of the improved algorithm. I show that one of the optimizations still creates a possibility for the algorithm to be incorrect. I also present a subclass of problems, for which the algorithm cannot decide reachability. I suggest new improvements that ensure correctness and extend the set of decidable problems.

The thesis is structured as follows. In Chapter 2 I present the necessary background

knowledge of formal modeling, Petri nets and reachability analysis. Chapter 3 gives an introduction to the general idea of the CEGAR approach and presents the CEGAR algorithm for Petri nets [1]. In Chapter 4 I examine the correctness and completeness of the algorithm and I suggest improvements. I also implemented the algorithm with the new contributions. Chapter 5 gives a brief insight to the implementation and the usage of the implemented tool. In Chapter 6 I present my measurement results on well-known models. Finally, I conclude my work in Chapter 7.

Chapter 2

Background

In this chapter I introduce the background knowledge required for understanding the topic of my thesis. At first I present Petri nets (Section 2.1.1) as the modeling formalism used in my work, and their extension with inhibitor arcs (Section 2.1.3). Then, I introduce the reachability analysis of Petri nets (Section 2.2) and the relevant parts of (Integer) Linear Programming (Section 2.3).

2.1 Petri nets

Petri nets are widely used for the modeling and analysis of asynchronous, parallel, distributed and non-deterministic systems [4], providing both structural and dynamical analysis techniques. Besides their mathematical formalism, they can also be represented graphically. Petri nets have various extensions (e.g., inhibitor arcs, priorities), which improve their modeling power, but the analysis techniques are usually limited for these extensions.

2.1.1 Definition

A Petri net is a directed, weighted bipartite graph. Nodes of the first set are called *transitions*, while nodes of the other set are called *places*. Each directed edge of the net connects a place and a transition and has a positive weight. Formally a Petri net is a tuple $PN = (P, T, E, W)$ [4], where:

$P = \{p_1, p_2, \dots, p_k\}$ is the finite set of places,

$T = \{t_1, t_2, \dots, t_n\}$ is the finite set of transitions,

$E \subseteq (P \times T) \cup (T \times P)$ is the set of edges,

$W : E \rightarrow \mathbb{Z}^+$ is the function assigning weights to the edges.

A state of the net can be described by a function $m : P \rightarrow \mathbb{N}$ assigning a non-negative integer to each place. This mapping is called the *marking* of the net. A place p is said to contain k *tokens* in a marking m if $m(p) = k$. The initial marking of the net is usually denoted by m_0 .

In the graphical representation of a Petri net, places are denoted by circles, transitions by rectangles and edges by arrows. If the weight of an edge is one, it is usually not labeled. The token distribution is denoted by numbers or black dots inside the places. An example

can be seen in Figure 2.1.

The input and output places of a transition $t \in T$ are denoted by $\bullet t$ and $t\bullet$, while the input and output transitions of a place $p \in P$ are denoted by $\bullet p$ and $p\bullet$, formally:

$$\begin{aligned}\bullet t &= \{p \mid (p, t) \in E\}, \\ t\bullet &= \{p \mid (t, p) \in E\}, \\ \bullet p &= \{t \mid (t, p) \in E\}, \\ p\bullet &= \{t \mid (p, t) \in E\}.\end{aligned}$$

Dynamic behavior

The dynamic behavior of Petri nets is defined by the *firing* (or *occurrence*) of transitions. The rules are the following:

- A transition $t \in T$ is *enabled* under a marking m , if at least $w^-(p, t)$ tokens are present in each of its input places $p \in \bullet t$, where $w^-(p, t)$ is the weight of the edge (p, t) . An enabled transition t under a marking m is denoted by $m[t]$.
- An enabled transition can fire, but it is not obligatory. If there are more than one enabled transitions, any of them can fire, which gives the ability for Petri nets to model non-determinism. The occurrence of a transition is an atomic event, so there are no “parallel” firings.
- When an enabled transition $t \in T$ fires, it consumes $w^-(p, t)$ tokens from its input places $p \in \bullet t$, and produces $w^+(p, t)$ tokens in its output places $p \in t\bullet$, where $w^+(p, t)$ is the weight of the edge (t, p) . The firing of the transition t under a marking m is denoted by $m[t]m'$, where m' is the marking after the occurrence of t .

The rules above hold for Petri nets without any extensions. The rules may be different for extended Petri nets and new rules can also be added (e.g., inhibitor arcs).

A word $\sigma \in T^n$ is called a *firing sequence*. A firing sequence is said to be *realizable* under a marking m and leads to m' , if a sequence of markings m_1, m_2, \dots, m_{n-1} exists, for which $m[\sigma(0)]m_1[\sigma(1)]m_2 \dots m_{n-1}[\sigma(n)]m'$ holds. This is denoted shortly by $m[\sigma]m'$. The *Parikh image* of a firing sequence σ is a vector $\wp(\sigma) : T \rightarrow \mathbb{N}$, where $\wp(\sigma)(t)$ is the number of occurrences of t in σ .

Example 1 A simple Petri net modeling a chemical process can be seen in Figure 2.1. The net has three places (H_2, O_2, H_2O) and two transitions (t_0, t_1). Figure 2.1(a) shows the initial marking, where only t_0 is enabled. If t_0 fires, the marking seen in Figure 2.1(b) is reached, where both t_0 and t_1 are enabled.

Reachability

A marking m' is *reachable* from a marking m if a realizable firing sequence $\sigma \in T^n$ exists, for which $m[\sigma]m'$ holds. The set of all reachable markings from the initial marking m_0 of a Petri net PN is denoted by $R(PN, m_0)$.

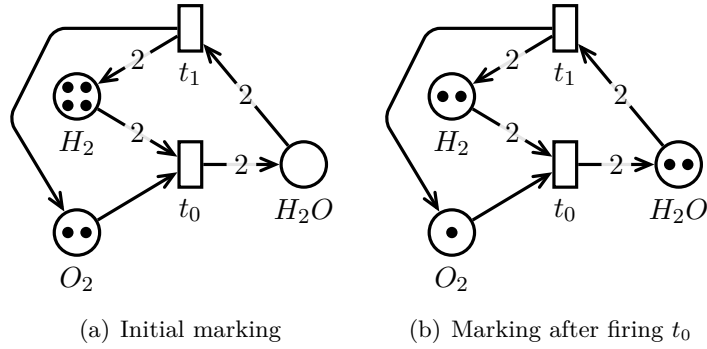


Figure 2.1: Example net modeling a chemical process

Boundedness

A Petri net is *bounded* if the number of tokens $m(p)$ is bounded for each place $p \in P$ and each reachable marking $m \in R(PN, m_0)$.

2.1.2 Incidence matrix and state equation

The *incidence matrix* of a Petri net $PN(P, T, E, W)$ is a matrix $C_{|P| \times |T|}$, where $C(i, j) = w^+(p_i, t_j) - w^-(p_i, t_j)$, i.e., the difference between the number of tokens in the place p_i after firing the transition t_j .

Example 2 The incidence matrix of the Petri net in Figure 2.2 is as follows.

$$\begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ -1 & -1 & 1 & 1 & -1 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix}$$

Note that the Petri net cannot always be restored from the incidence matrix, e.g., the two edges between t_0 and p_2 appear as a zero in the matrix.

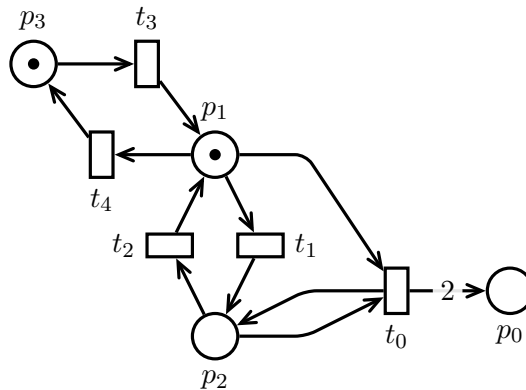


Figure 2.2: Example Petri net

A marking m of the Petri net $PN = (P, T, E, W)$ can be written as a column vector, where the i th element is the token count of the place $p_i \in P$. The *firing vector* u of a

transition t_j is a column vector filled with zeros, except for the j th place, which is one. Due to the definition of the incidence matrix, if t_j is enabled under a marking m , the marking m' after firing t_j ($m[t_j)m'$) can be obtained using the following equation:

$$m + Cu = m'.$$

This can be generalized for a firing sequence $\sigma \in T^n$, where the firing vectors of the transitions are represented by u_1, u_2, \dots, u_n :

$$m + Cu_1 + Cu_2 + \dots + Cu_n = m'.$$

Since matrix-vector multiplication is distributive, substituting $x = u_1 + u_2 + \dots + u_n$ into the previous equation yields the following equation, which is called the *state equation*:

$$m + Cx = m'.$$

The j th component of x is the number of occurrences of the transition t_j in σ . A vector $x \in \mathbb{N}^{|T|}$ fulfilling the state equation is called a *solution*. Note that for any realizable firing sequence σ with $m[\sigma)m'$, the Parikh image $\wp(\sigma)$ fulfills the state equation:

$$m + C\wp(\sigma) = m'.$$

On the other hand, not every solution of the state equation is a Parikh image of a realizable firing sequence. A solution x is called *realizable* if a realizable firing sequence σ exists, with $\wp(\sigma) = x$.

Example 3 Consider the Petri net in Figure 2.3(a) with $m_1 = (0)$ and $m'_1 = (1)$. The vector $x_1 = (1)$ fulfills the state equation, but firing t_0 is not possible. Now consider the Petri net in Figure 2.3(b) with $m_2 = (2, 0)$ and $m'_2 = (0, 4)$. The vector $x_2 = (2)$ fulfills the state equation, and firing t_0 two times is possible.

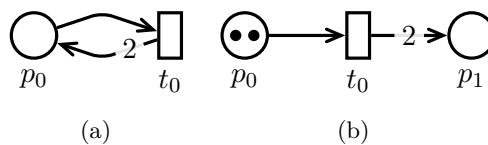


Figure 2.3: Example for an unrealizable and a realizable solution

T-invariants

A vector $x \in \mathbb{N}^{|T|}$ is a *T-invariant* if $Cx = 0$ holds. The firing of a T-invariant does not change the marking, since $m + Cx = m + 0 = m$. A T-invariant x is called *realizable* if a realizable firing sequence σ exists, with $\wp(\sigma) = x$. T-invariants usually represent the possibility of a cyclic behavior in the modeled system. The sum of two realizable T-invariants is also realizable (by firing them after each other), but a realizable T-invariant

cannot always be split into a sum of two realizable T-invariants.

Example 4 [1] *The Petri net in Figure 2.4 has three invariants¹: $T_1 = \{t_0, t_1\}$, $T_2 = \{t_2, t_3\}$ and $T_3 = \{t_0, t_1, t_2, t_3\}$ with $T_3 = T_1 + T_2$. T_3 can be realized by the firing sequence (t_0, t_2, t_3, t_1) , but T_2 is not realizable, since both t_2 and t_3 are disabled.*

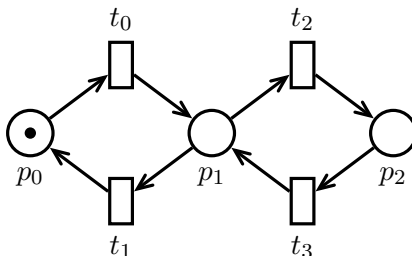


Figure 2.4: Example for T-invariants

2.1.3 Inhibitor arcs

Petri nets have several extensions to improve their modeling power, including *inhibitor arcs*, prioritized transitions and self-modifying nets [5], from which I use inhibitor arcs in my work. With inhibitor arcs, it is possible to model that some action of the modeled system cannot happen under some circumstances. The set of inhibitor arcs is denoted by $I \subseteq (P \times T)$ and a Petri net extended with inhibitor arcs is a tuple $PN_I = (PN, I)$. There is an extra rule for a transition t to be enabled using inhibitor arcs: each place connected to a transition with an inhibitor arc must have zero tokens. Formally, a transition $t \in T$ is enabled in a Petri net PN_I under a marking m if t is enabled in PN and for each $p \in P$, if $(p, t) \in I$, then $m(p) = 0$ must hold. In the graphical representation, inhibitor arcs have a small circle instead of the arrowhead (Figure 2.5).

Example 5 *An example net containing inhibitor arcs can be seen in Figure 2.5. Figure 2.5(a) shows the initial marking, where t_0 is disabled by the inhibitor arc connecting t_0 to p_0 . After firing t_1 (Figure 2.5(b)) p_0 has zero tokens, so t_0 can now fire. The marking in Figure 2.5(c) is reached after firing t_0 .*

Petri nets extended with inhibitor arcs are *Turing complete*, but also have some disadvantages. Inhibitor arcs do not appear in any form in the state equation. Therefore, a solution of the state equation can be unrealizable due to an inhibitor arc. A Petri net PN_I containing inhibitor arcs can be transformed into an equivalent PN net, but this method only works for bounded nets [6]. Analysis methods are also limited when using inhibitor arcs [7], e.g., the reachability analysis (Section 2.2) is undecidable in general [5].

¹When a T-invariant x contains only zeros and ones, I denote x by the transitions corresponding to the ones, e.g., $(1, 0, 1, 0) = \{t_0, t_2\}$

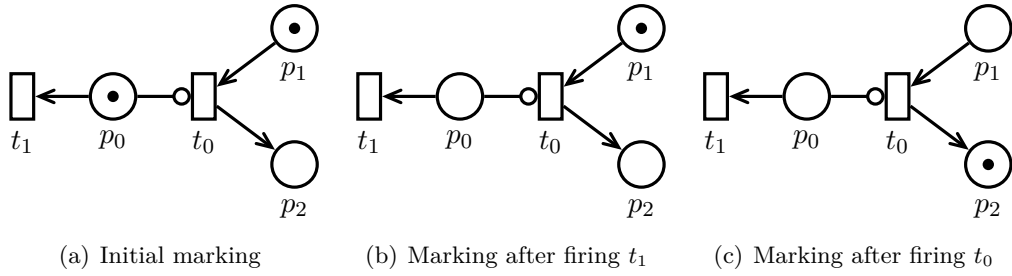


Figure 2.5: Example net with inhibitor arcs

2.2 Reachability analysis

One of the most important formal verification technique for Petri nets is the reachability analysis. The aim of the reachability analysis is to check if a given marking is reachable from the initial marking. A generalization of reachability is the submarking coverability analysis, where linear conditions are given instead of the target marking.

2.2.1 Reachability problem

The *reachability problem* is a tuple (PN, m_0, m') , where m_0 is the initial and m' is the target marking². The answer to the reachability problem is “yes” if and only if $m' \in R(PN, m_0)$, i.e., m' can be reached from m by some realizable firing sequence.

2.2.2 Submarking coverability problem

The reachability of an exact target state may not be general enough. We may want to give linear conditions on the marking to be reached. These conditions are called *predicates*, and have the following form [8]:

$$\mathcal{P}(m) \Leftrightarrow Am \geq b,$$

where $\mathcal{P}(m)$ represents the predicate, A is a matrix and b is a vector of coefficients. The predicate $\mathcal{P}(m)$ holds if and only if $Am \geq b$ holds. The answer to the submarking coverability problem (PN, m_0, \mathcal{P}) is “yes” if and only if a marking m' can be reached from m_0 , for which $\mathcal{P}(m')$ holds.

Example 6 Consider the Petri net in Figure 2.6 and suppose that we want to reach a marking m with $m(p_0) = 2$. The firing sequence (t_1, t_3, t_0) fulfills this criterion, but the marking also changes in some of the other places. Using reachability analysis we must give the marking of the other places as well, which we may not know in advance. Using submarking coverability we can define only the marking of p_0 .

Example 7 Consider the Petri net in Figure 2.6 with the question whether we can have two tokens in p_0 while the sum of the tokens in p_1 and p_3 must be at least one. The reachability analysis cannot answer this, but using submarking coverability we obtain the firing sequence (t_1, t_3, t_0, t_2) as a solution.

²In my thesis I also use the notation $m_0 \rightarrow m'$ for the reachability problem.

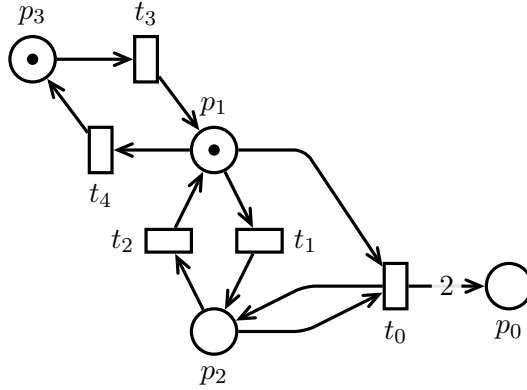


Figure 2.6: Submarking coverability example

2.2.3 Necessary and sufficient criteria

Both the reachability and the submarking coverability problems have necessary and sufficient criteria.

Reachability problem

If the answer to the reachability problem $m' \in R(PN, m_0)$ is “yes”, then m' is reachable from m_0 by some firing sequence σ . The Parikh image $\wp(\sigma)$ fulfills $m_0 + C\wp(\sigma) = m'$, therefore the feasibility of the state equation is a necessary criterion for reachability. On the other hand, this criterion is not sufficient, the Petri net in Figure 2.3(a) is a counterexample. A sufficient criterion for reachability is showing a realizable firing sequence σ , with $m_0[\sigma]m'$.

Submarking coverability

Similarly to the reachability problem, the feasibility of the state equation $m_0 + Cx = m'$ and the predicate $Am' \geq b$ is a necessary, but not sufficient criterion for submarking coverability. A realizable firing sequence σ with $m_0[\sigma]m'$ and $Am' \geq b$ is a sufficient criterion.

2.2.4 Decidability and complexity

The reachability analysis of Petri nets belongs to the hard problems of mathematics. It is decidable [9], but no upper bound of its complexity is known yet. R. J. Lipton proved that the problem is at least EXPSPACE-hard [10]. This means that no algorithm can solve the problem efficiently in the general case.

Petri nets containing inhibitor arcs can be transformed into equivalent PN nets by splitting places, but this method only works for bounded nets [6]. It was proved (using the 10th problem of Hilbert) that in general, reachability is undecidable for Petri nets extended with inhibitor arcs [5].

2.2.5 Existing methods for solving reachability

There are several methods for solving the reachability problem, each having its advantages and disadvantages.

Naive approach

The most naive approach is to traverse the state space explicitly by breadth or depth first search. This approach is useless for practical problems, since the state space can be very large or even infinite³. Also, this method fails for proving that a marking is not reachable in an infinite state space.

Symbolic algorithms

Symbolic representations try to overcome the state explosion problem by storing the states in a compact, encoded form. Saturation [11] is a very effective iteration strategy realizing this approach. However, it only works on finite state spaces, and can be slow if the target marking is “far” from the initial marking.

Abstraction

The subject of my thesis is an algorithm that uses abstraction (Chapter 3). The state space of the abstract model has a finite representation and it is usually less complex. However, a state reached in the abstract model may not be reachable in the original one, due to some details hidden by the abstraction. In such cases, the abstraction needs to be refined. The major drawback of abstraction is that the completeness and correctness of the algorithm is not trivial.

2.3 Linear programming (LP)

Solving the state equation and finding markings satisfying a predicate is a *linear programming* problem. Linear programming is a mathematical approach for finding an optimal solution in a given mathematical model and requirements [12]. Formally, the purpose of linear programming is to minimize an objective function, subject to linear equalities and linear inequalities. The canonical form of a linear programming problem is the follows:

$$\begin{aligned} & \text{minimize} && c^T x, \\ & \text{subject to} && Ax \leq b \text{ and } x \geq 0, \end{aligned}$$

where x is the vector of variables, b , c are vectors and A is a matrix of coefficients. The feasible region of the linear programming problem is a convex polyhedron, which is determined as the intersection of finite number of half spaces. The aim of the linear programming problem is to determine the point in the feasible region, where the objective function is minimal (or maximal), if such point exists. Several algorithms can solve the linear programming problem efficiently in polynomial time.

³This problem is often called the “state explosion” problem.

2.3.1 Integer Linear Programming (ILP)

When all the variables are integers, the problem is called *integer linear programming* problem. Despite linear programming, integer linear programming is an NP-hard problem. Finding solutions for the state equation of Petri nets is an integer linear programming problem, since the firing count of a transition must be integer.

Chapter 3

CEGAR approach on Petri nets

In this chapter I introduce the basic concept of **C**ounter**E**xample **G**uided **A**bstraction **R**efinement (CEGAR) and its application on the reachability problem of Petri nets [1]. Section 3.1 gives a short introduction on the general idea of CEGAR, while Section 3.2 presents the details of the CEGAR approach on the reachability problem.

3.1 The CEGAR approach

A major drawback of formal modeling and analysis is their high computational complexity. Even for relatively small models the state space can be very large or infinite. Counterexample guided abstraction refinement is a general approach, which can handle large and infinite state spaces. The flowchart of the CEGAR approach can be seen in Figure 3.1. The details of the process are explained in the following subsections.

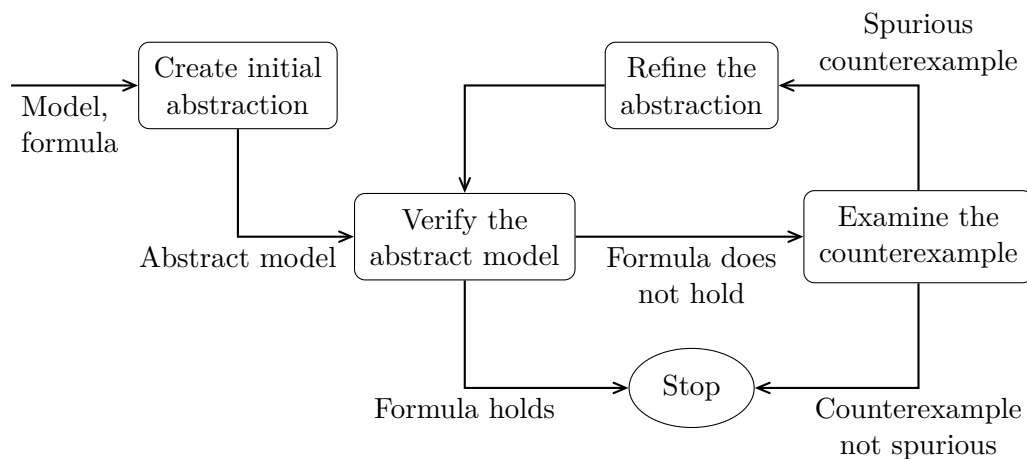


Figure 3.1: CEGAR flowchart

3.1.1 Abstraction

Abstraction is a mathematical approach, which is widely used to solve hard problems. It hides the irrelevant details, so the abstract model can be handled easier than the original one. If any important detail is lost, the abstraction has to be refined. There are several

types of abstraction methods, both for proving and disproving the feasibility of a given condition. The general CEGAR approach uses existential abstraction, which means that the abstract model is an over-approximation of the original one, i.e., the abstract model has more behaviors, while no behavior of the original model is lost. Using existential abstraction, a group of states in the original model is mapped to a single state of the abstract model.

3.1.2 Counterexample guided refinement

The first step of the CEGAR approach is to create an initial abstraction using the model and the formula. The next step is to verify the abstract model. If the formula holds in the abstract model, it also holds in the original model due to the over-approximation. If there is a counterexample, for which the formula does not hold, it might be caused by the over-approximation of the abstraction. In this case the counterexample is examined, whether it is also a counterexample in the original model.

If the counterexample is not spurious, the formula does not hold in the original model. Otherwise, the abstraction has to be refined to exclude the previous counterexample, and the verification is repeated.

3.2 Reachability analysis of Petri nets using CEGAR

This section introduces a recently published algorithm [1], which applies the CEGAR approach on the reachability problem of Petri nets. The flowchart of the Petri net specific CEGAR approach can be seen in Figure 3.2.

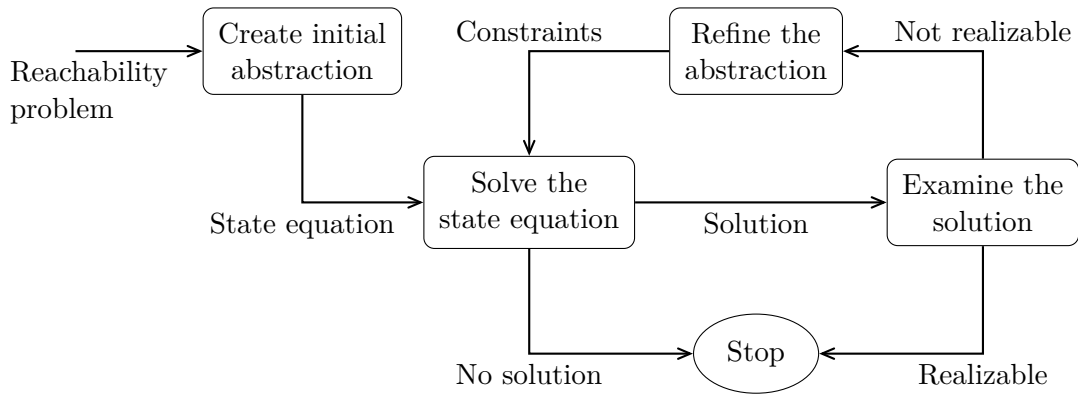


Figure 3.2: *Petri net specific CEGAR flowchart*

The initial abstraction of the reachability problem is the state equation (Section 2.1.2). The feasibility of the state equation is a necessary, but not sufficient criterion for reachability. Therefore, it is an over-approximating abstraction of the reachability problem. The abstract model can be verified by solving the state equation, which is an integer linear programming problem. The ILP solver tool can yield one solution, optimizing a linear function. Each coefficient of the optimized function is one, so the shortest solutions are

produced¹. Trying to solve the state equation, the following cases are possible:

- If the state equation is infeasible, the target marking is not reachable, since the necessary criterion does not hold.
- If a solution x exists, it must be examined, whether there is a realizable firing sequence σ with $\wp(\sigma) = x$.
 - If such firing sequence σ exists, the target marking is reachable.
 - If no realizable firing sequence can be found, than the solution x is a counterexample, and the abstraction has to be refined.

The purpose of the abstraction refinement is to exclude the counterexample from the solution space without losing any realizable solution. This can be achieved by adding additional linear inequalities (*constraints*) to the state equation. Due to the new constraints, the state equation either becomes infeasible, or a different solution is produced by the ILP solver. The following section introduces the form of the solution space and the constraints used by the algorithm.

3.2.1 Solution space of the state equation

Each solution vector x fulfilling the state equation $m_0 + Cx = m'$ can be written as the sum of a *base vector* and the linear combination of T-invariants [1]. Formally, $x = b_n + \sum_i n_i y_i$, where b_n is a vector from a finite set of pairwise incomparable² base vectors and y_i is a minimal T-invariant with the coefficient $n_i \in \mathbb{N}$.

Constraints

Two types of constraints were defined in [1]:

- *Jump constraints* have the form $|t_i| < n$, where $t_i \in T$, $n \in \mathbb{N}$ and $|t_i|$ represents the firing count of the transition t_i . Jump constraints can be used to switch between different base vectors, exploiting the fact that they are pairwise incomparable.
- *Increment constraints* have the form $\sum_{t_i \in T} n_i |t_i| \geq n$, where $n_i \in \mathbb{Z}$ and $n \in \mathbb{N}$. Increment constraints can be used to reach non-base solutions, i.e., involving T-invariants.

Example 8 Consider the Petri net in Figure 3.3(a) with the reachability problem $(0, 0, 1, 0) \rightarrow (1, 0, 1, 0)$. There are two base vectors for this problem: $(1, 0, 0)$ (firing t_0) and $(0, 1, 1)$ (firing t_1 and t_2). Since the ILP solver minimizes the firing count of transitions, it yields the solution $(1, 0, 0)$ first, which is unrealizable. Using a jump constraint $|t_0| < 1$, the ILP solver can be forced to produce the realizable solution $(0, 1, 1)$.

¹In my implementation, the coefficients can be set manually to arbitrary values.

²Two vectors are pairwise incomparable if both have a component that is lesser than the same component of the other vector, e.g., $(2, 0, 1)$ and $(1, 2, 1)$.

Example 9 Consider the Petri net in Figure 3.3(b) with the reachability problem $(0, 0, 1) \rightarrow (1, 0, 1)$. The only base vector for this problem is the vector $(1, 0, 0)$ (firing t_0), which is unrealizable. Using an increment constraint $|t_1| \geq 1$, the ILP solver can be forced to add the T-invariant $\{t_1, t_2\}$ to the new solution $(1, 1, 1)$, which is realizable by the firing sequence $\sigma = (t_1, t_0, t_2)$.

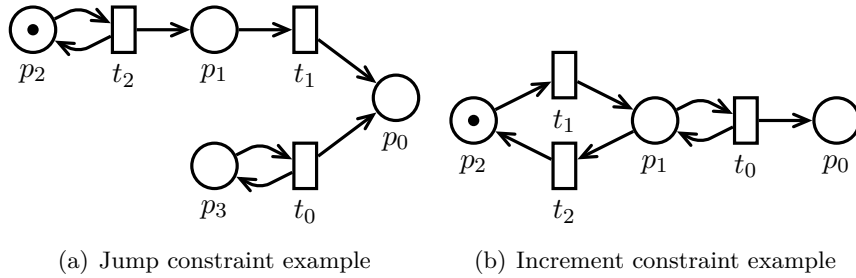


Figure 3.3: Example nets for jump and increment constraints

Figure 3.4 shows a visualization of the solution space [1]. Black dots represent solution vectors, while cones represent the linear space formed by the T-invariants. Jumps are denoted by dashed arcs and increments are denoted by normal arcs. The base solutions are the ones on the bottom level. Note that jumps can occur on higher levels (dotted arc), and can be used to switch between different T-invariants as well.

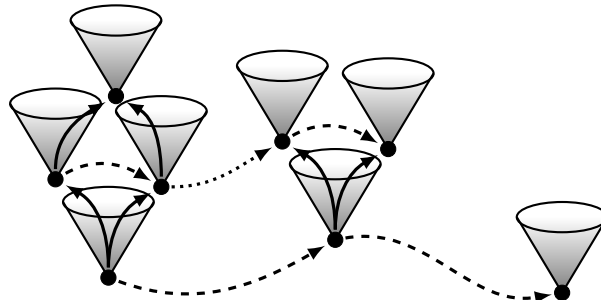


Figure 3.4: Solution space of the state equation

Section 3.2.3 explains how to create jump and increment constraints in order to reach a realizable solution, but first the concept of *partial solutions* is introduced in the following section.

3.2.2 Partial solutions

Partial solutions are generated from a solution of the state equation (and constraints) by firing as many transitions of the solution vector as possible. Formally, a partial solution of a Petri net $PN = (P, T, E, W)$ and a reachability problem $m' \in R(PN, m_0)$ is a tuple $(\mathcal{C}, x, \sigma, r)$, where:

- \mathcal{C} is the set of jump and increment constraints that are added to the state equation,

- $x \in \mathbb{N}^{|T|}$ is the minimal³ solution satisfying both the state equation and the constraints of \mathcal{C} ,
- $\sigma \in T^*$ is a maximal realizable firing sequence, with $\wp(\sigma) \leq x$ (i.e., each transition can fire at most as many times as it is included in the solution vector),
- $r = x - \wp(\sigma)$ is the remainder vector.

Since the firing sequence σ has to be maximal, all transitions t_i with $r(t_i) > 0$ are disabled after firing σ .

Generating partial solutions

Partial solutions are obtained from a solution x and a constraint set \mathcal{C} by firing as many transitions as possible. The algorithm uses a “brute force” approach for this purpose: it builds a tree with markings as nodes and transitions as edges. The root of the tree is the initial marking m_0 and an edge labeled t_i is present between the nodes m' and m'' if $m'[t_i]m''$ holds. On each path leading from the root of the tree to a leaf, every transition t_i can occur at most $x(t_i)$ times. Each path to a leaf represents a maximal firing sequence σ , therefore a partial solution $(\mathcal{C}, x, \sigma, r)$. Although the tree of the partial solutions can be traversed only storing one path in the memory at a time, the size of the tree can grow exponentially. Some optimization methods are presented in Section 3.2.5 to reduce the branching factor of the tree.

Example 10 Consider the Petri net in Figure 3.5(a) with the solution vector $x = (2, 1)$. The tree of partial solutions for x can be seen in Figure 3.5(b). There are three different maximal firing sequences, thus three partial solutions. Note that the length of the firing sequences can be different.

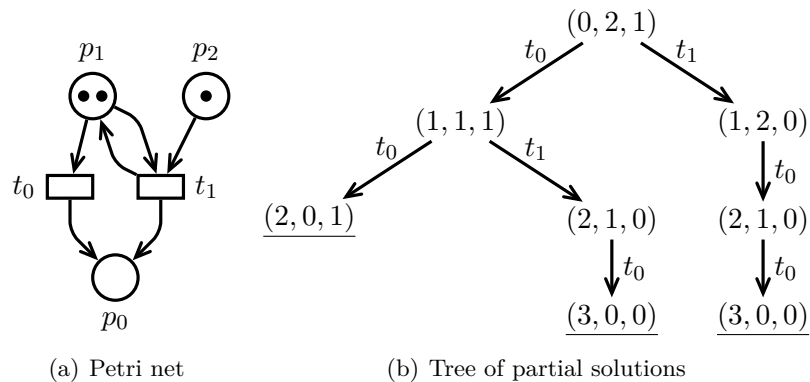


Figure 3.5: Partial solution tree example

³The vector x is minimal regarding the optimized function of the ILP solver.

Examination of partial solutions

A partial solution is called a *full solution* if $r = 0$ holds. A full solution is a sufficient criterion for reachability, since if $r = 0$, then $\wp(\sigma) = x$, which means that x is realizable by the firing sequence σ .

We know that for each realizable solution x of the state equation, a full solution $(\{c\}, x, \sigma, 0)$ exists, where $\wp(\sigma) = x$ and c is a constraint of the form $\sum_{t_i \in T} x(t_i) |t_i| \geq \sum_{t_i \in T} x(t_i)$, which ensures that x is the minimal solution produced by the ILP solver [1]. Furthermore, full solutions can also be reached by continuously expanding the initial (minimal) solution with (jump and increment) constraints [1].

Consider now a partial solution $PS = (\mathcal{C}, x, \sigma, r)$, which is not a full solution, i.e., $r \neq 0$. This means that some transitions in the solution vector x could not fire enough times in the firing sequence σ . The following cases are possible in this situation:

1. x may be realizable by another firing sequence σ' , thus a full solution $PS' = (\mathcal{C}, x, \sigma', 0)$ can be found on an alternate path of the partial solution tree for x .
2. Greater, but pairwise incomparable solution vectors can be obtained using jump constraints.
3. Increment constraints can be added to increase the token count in the input places of transitions $t_i \in T$ with $r(t_i) > 0$. Since the final marking must not change, this can be achieved by adding T-invariants to the solution x . These invariants can “lend” tokens in the input places of transitions with $r(t_i) > 0$.

The following section introduces methods for generating jump and increment constraints in order to reach a full solution.

3.2.3 Generating constraints

When a partial solution is not a full solution, both jump and increment constraints can be added, but they are applied on a different level of the solution space:

- Jump constraints are generated from solution vectors of the state equation.
- Increment constraints are generated from partial solutions (which were obtained from solution vectors).

Jump constraints

Given a solution vector x , jump constraints are used to obtain new, pairwise incomparable solution vectors. For each transition $t_i \in T$ with $x(t_i) > 0$ a jump constraint c_i of the form $c_i = |t_i| < x(t_i)$ can be added to the state equation. If a new solution vector y_i is obtained after adding one of the constraints c_i , this process can be recursively repeated for y_i . We know that every base solution can be obtained using jump constraints [1].

Reaching non-base solutions require increment constraints, but they might contradict with previous jump constraints. As an example, consider that the solution $b_1 = (2, 0, 0)$ is

reached with the jump constraint $|t_1| < 1$ from the minimal solution $b_0 = (0, 1, 0)$. If now we want to add the T-invariant $\{t_1, t_2\}$ to b_1 three times that would contradict with the previous constraint. Since jumps are only used to obtain pairwise incomparable solutions, they can be transformed into equivalent increment constraints using the following method.

Transforming jumps

Suppose that \mathcal{C} is a set of jump and increment constraints and z is the minimal solution fulfilling the state equation and \mathcal{C} . Let \mathcal{C}' include each increment constraint of \mathcal{C} and an additional increment constraint of the form $|t_i| > z(t_i)$ for each transition $t_i \in T$. Then, a vector $y \geq z$ is a solution of the state equation plus $\mathcal{C} \cap \mathcal{C}'$ if and only if y is a solution of the state equation and \mathcal{C}' [1]. Furthermore, no solution smaller than z fulfills the state equation and \mathcal{C}' [1]. This means that if we are interested in the solutions of the cone over z , we can replace \mathcal{C} with \mathcal{C}' , which no longer contains jump constraints.

Increment constraints

Consider a partial solution $PS = (\mathcal{C}, x, \sigma, r)$, which is not full, i.e., some transitions could not fire enough times. In this case, increment constraints can be used to extend the solution vector x with T-invariants, which may help enabling transitions in the remainder r . In order to get the proper invariants, we must determine the places that require additional tokens and the number of required tokens. This problem is harder than testing if a solution is realizable, i.e., testing if zero additional tokens are enough. A recursive approach would be ineffective, since many different remainder vectors may belong to a solution vector. Although the remainder is smaller than x , the number of recursion steps might grow exponentially with the size of x [1].

The authors of the algorithm [1] use a heuristic to find the places and the required number of tokens to enable transitions that could not fire. If a set of places actually require n ($n > 0$) additional tokens, the heuristic gives an estimation between 1 and n . If the estimation is too low, the heuristic can be applied again converging to n . The heuristic consists of the following three steps:

1. Build a dependency graph between places and transitions to determine the set of places that require additional tokens.
2. Calculate the number of tokens needed for each set of places.
3. Generate a constraint for each set of places using the information from the previous steps.

The first step is to build a dependency graph known from partial order reduction methods [13]. Let \hat{m} be the marking reached after firing σ of the partial solution $PS = (\mathcal{C}, x, \sigma, r)$, i.e., $m_0[\sigma]\hat{m}$. The dependency graph $G = (P_0 \cup T_0, E)$ can be constructed the following way:

- $T_0 = \{t \in T \mid r(t) > 0\}$.

- $P_0 = \{p \in P \mid \exists t \in T_0 : w^-(p, t) > \hat{m}(p)\}$.
- $E = \{(p, t) \in P_0 \times T_0 \mid w^-(p, t) > \hat{m}(p)\} \cup \{(t, p) \in T_0 \times P_0 \mid w^+(p, t) > w^-(p, t)\}$.

The graph consists of transitions that could not fire enough times (T_0) and places (P_0) that disable these transitions under the marking \hat{m} . The edges of the graph have a different meaning regarding their direction. An edge (p, t) means that p disables t under \hat{m} , while an edge (t, p) means that firing t would increase the token count of p .

A strongly connected component (SCC) of the graph represents a set of transitions that could mutually enable each other if some places of the SCC got additional tokens. Consider now a source SCC, i.e., one with no incoming edges. The token requirement of this SCC cannot be fulfilled by other SCCs, therefore transitions outside the remainder ($t \in T$ with $r(t) = 0$) must fire. For each source SCC the algorithm determines a tuple (P_i, T_i, X_i) , where:

- $P_i = SCC \cap P_0$ is the set of places of the SCC,
- $T_i = SCC \cap T_0$ is the set of transition of the SCC,
- $X_i = \{t \in T_0 \setminus SCC \mid \exists p \in P_i : (p, t) \in E\}$ is the set of transitions outside the SCC that depend on the actual SCC.

The second step is to calculate the token requirement of each tuple (P_i, T_i, X_i) . The exact number is hard to determine, since a transition in T_i may enable all the other transitions in $T_i \cup X_i$. The following heuristic [1] gives a good estimation for a tuple (P_i, T_i, X_i) :

- If $T_i \neq \emptyset$, then enabling a transition $t_i \in T_i$ may enable all the other transitions, since t_i can produce additional tokens in some places of P_i . In this case n is the number of tokens required by the transition missing the least tokens, which is formally $n = \min_{t \in T_i} (\sum_{p \in P_i} \max(0, w^-(p, t) - \hat{m}(p)))$.
- If $T_i = \emptyset$, then P_i can only consist of a single place p_i . This means that the token requirements of all the transitions in X_i must be fulfilled by p_i . The transitions of X_i can also produce tokens in p_i , which has to be considered in the estimation. Transitions therefore, are ordered in groups. Each group G_j consists of transitions $t \in X_i$ with $w^+(p_i, t) = j$. Firing the transitions of a group with the smallest value j last minimizes the leftover at p_i . Transitions in the same group G_j can be processed at once, each using $w^-(p_i, t) - j$ tokens, except for the first one, which requires j additional tokens. The tokens produced by a group G_j can be consumed by the next group G_{j-1} . After processing each group, we get the estimated number of tokens required to fire each transition in X_i .

The third step is to calculate an increment constraint. Let P_i be a set of places, which require n additional tokens and $T_i = \{t \in T \mid r(t) = 0 \wedge \sum_{p \in P_i} (w^+(p, t) - w^-(p, t)) > 0\}$, i.e., transitions outside the remainder vector that can produce tokens in the places of P_i .

In this case the increment constraint c has the following form [1]:

$$\sum_{t \in T_i} \sum_{p \in P_i} (w^+(p, t) - w^-(p, t)) \cdot |t| \geq n + \sum_{t \in T_i} \sum_{p \in P_i} (w^+(p, t) - w^-(p, t)) \cdot \wp(\sigma)(t).$$

The left-hand side consists of transitions weighted with the number of tokens produced in P_i . The formula on the right-hand side is the number of tokens already produced by the transitions in the firing sequence σ .

The constraint c can now be added to the set of constraints \mathcal{C} , forcing the ILP solver to produce a solution $x + y$ (with y being an invariant).

3.2.4 Reachability of solutions

The following theorem [1] states that if the reachability problem has a solution, it can be reached by the CEGAR method:

Theorem 1 *If the reachability problem has a solution, a realizable solution of the state equation can be reached by continuously expanding the minimal solution with jump and increment constraints.*

3.2.5 Optimizations

Some optimization methods were also presented in [1]. These methods reduce the search space and can also prevent non-termination.

Stubborn set

The size of the partial solution tree can grow exponentially with the number of transitions in the solution vector. There are several *partial order reduction* methods that can reduce the branching factor of this tree. One of them is the so-called *stubborn set* method [14], which is used by the CEGAR algorithm.

The stubborn set method investigates conflicts and dependencies between transitions and cuts the state space without losing any state of interest. At each reached marking m , a set of transitions, called the stubborn set ($stub(m)$) is determined, which is a subset of the enabled transitions at m . The state space exploration continues from m only with transitions $t \in stub(m)$. This way, the size of the state space is smaller than continuing with each enabled transition.

There are several stubborn set definitions for different properties investigated. The CEGAR algorithm should not lose any reachable state, for which the following definitions must hold [14]:

- **D1:** If $t \in stub(m)$, $t_1, t_2, \dots, t_n \notin stub(m)$, $m[t_1 t_2 \dots t_n] m_n$ and $m_n[t] m'_n$, then there exists a marking m' such that $m[t] m'$ and $m'[t_1 t_2 \dots t_n] m'_n$.
- **D2:** If m has an enabled transition, then there must be at least one transition $t_k \in stub(m)$ such that if $t_1, t_2, \dots, t_n \notin stub(m)$ and $m[t_1 t_2 \dots t_n] m_n$ then $m_n[t_k]$. Each transition with this property is called a *key transition* of $stub(m)$.

An algorithm is presented in [2] for determining the stubborn set at a marking m , satisfying the previous criteria.

Subtree omission

When a transition t of the solution vector x has to fire more than once (i.e., $x(t) > 1$), the stubborn set method alone is not efficient. The same state is often reached by firing sequences only different in the order of transitions. An example can be seen in Figure 3.6. Suppose that a marking \tilde{m} is reached from m_0 (by a firing sequence α), where both transitions t and u are enabled. Furthermore, suppose that a marking \hat{m} is reachable by two firing sequences $t\sigma u$ and $u\sigma t$ only different in the order of transitions. If the subtree after $\alpha t\sigma u$ is already processed, then the subtree after $\alpha u\sigma t$ can be omitted, since the order of the transitions does not matter for the abstraction refinement.

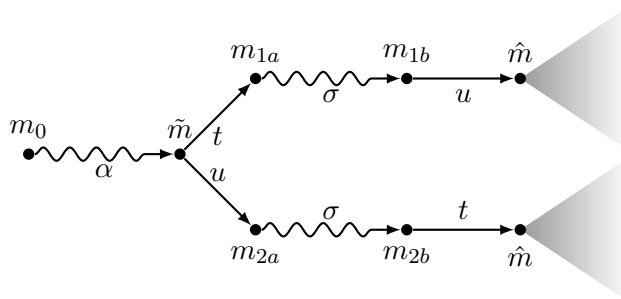


Figure 3.6: If $\alpha t\sigma u$ and $\alpha u\sigma t$ can both fire, then only one of the subtrees after \hat{m} needs to be processed.

Filtering T-invariants

Suppose that the abstraction refinement adds the T-invariant y to the solution vector x of a partial solution $PS = (\mathcal{C}, x, \sigma, r)$ in order to enable transitions with $r(t) > 0$. If y can be fired without enabling any transition in r , a partial solution $PS' = (\mathcal{C}', x + y, \sigma', r)$ is found. This means that the algorithm did not get any closer to finding a full solution by PS' : the same transitions have to fire under the same marking, like in PS . The abstraction refinement therefore, adds y again, which can prevent the algorithm from terminating. Such situations should be detected, and PS' has to be skipped. However, it is possible that at some marking during the firing of y , the algorithm was closer to enabling a transition t with $r(t) > 0$, i.e., less tokens were missing from the input places of t at an intermediate marking than at the final marking. These intermediate markings should be detected and used as new partial solutions, otherwise full solutions can be lost.

Storing partial solutions

Some partial solutions may occur many times (e.g., by adding the same constraints in a different order). As long as we have enough memory, partial solutions should be stored to avoid doing the abstraction refinement multiple times.

3.2.6 A complex example

This section presents a complex example, which shows how the algorithm traverses the solution space, including the T-invariant filtering optimization. The Petri net can be seen in Figure 3.7 and the reachability problem is $(0, 0, 0, 2) \rightarrow (1, 0, 0, 2)$, i.e., to produce a token in p_0 . The solution space is presented in Figure 3.9 and it is explained thoroughly in this section.

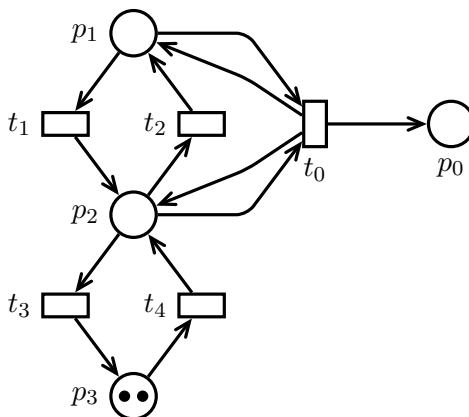


Figure 3.7: A complex example showing several aspects of the algorithm

The root of the solution space is the minimal solution vector $(1, 0, 0, 0, 0)$, denoted by SV_0 (i.e., firing t_0). Since t_0 is not enabled, the only partial solution is PS_0 , with the empty firing sequence σ_0 . The algorithm builds a dependency graph (Figure 3.8(a)) to determine the increment constraints. The graph has edges from p_1 and p_2 to t_0 because they disable t_0 . Edges in the opposite direction are not present, since firing t_0 does not increase the token count of p_1 or p_2 .

There are two source SCCs for PS_0 :

- $SCC_1(\{p_1\}, \emptyset, \{t_0\})$: One token is required in p_1 , where t_2 can produce tokens, so the constraint is $|t_2| \geq 1$.
- $SCC_2(\{p_2\}, \emptyset, \{t_0\})$: One token is required in p_2 , where t_1 and t_4 can produce tokens, so the constraint is $|t_1| + |t_4| \geq 1$.

The new minimal solution fulfilling the state equation and the constraints is $(1, 1, 1, 0, 0)$, labeled by SV_1 (i.e., the T-invariant $\{t_1, t_2\}$ is added). Since none of the transitions t_0, t_1, t_2 is enabled, the only partial solution is PS_1 with the empty firing sequence σ_1 . The dependency graph for PS_1 can be seen in Figure 3.8(b).

There are two edges going from transitions to places as well, since t_1 and t_2 can increase the token count of p_2 and p_1 . The only source SCC is $SCC(\{p_1, p_2\}, \{t_1, t_2\}, \{t_0\})$. One token in p_1 or p_2 might enable all the transitions of the SCC. The increment constraint takes the form $|t_4| \geq 1$, since t_4 is the only transition outside the remainder that can produce tokens in the SCC.

The new solution vector is $(1, 1, 1, 1, 1)$, denoted by SV_2 (i.e., the T-invariant $\{t_3, t_4\}$ is added). Two partial solutions can be found for SV_2 :

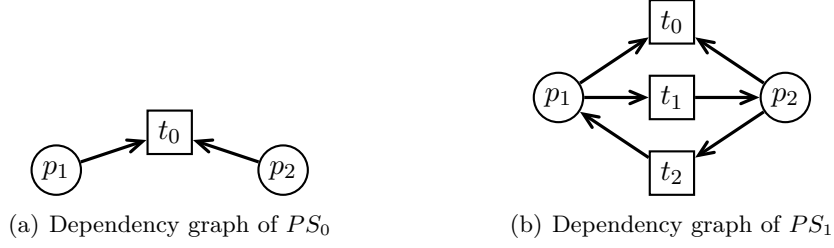


Figure 3.8: *Dependency graphs of PS_0 and PS_1*

- PS_{21} has the firing sequence $\sigma_{21} = (t_4, t_3)$, but it is skipped by the T-invariant filtering optimization: it has the same remainder as PS_1 and the firing sequences are only different in the T-invariant $\{t_3, t_4\}$ ⁴. However, if only t_4 is fired from $\sigma_{21} = (t_4, t_3)$, we are closer to enabling t_0 , since it misses only one token. This better intermediate state is denoted by BS_1 . In BS_1 , one token is missing from p_1 , where only t_2 could produce tokens, but $r(t_2) > 0$, so this partial solution cannot be extended with constraints.
- The other partial solution is PS_{22} with the firing sequence $\sigma_{22} = (t_4, t_2, t_1, t_3)$. PS_{22} is also skipped by the T-invariant filtering optimization: it has the same remainder as PS_0 and the firing sequences are only different in the T-invariant $\{t_1, t_2, t_3, t_4\}$. However, there is a better intermediate state BS_2 , where only t_4 and t_2 is fired from σ_{22} . This intermediate state misses a token from p_2 , where t_1 and t_4 can produce tokens. Since $r(t_1) > 0$, the constraint is $|t_4| \geq 2$.

The new solution vector is $(1, 1, 1, 2, 2)$, denoted by SV_3 (i.e., the T-invariant $\{t_3, t_4\}$ is added). SV_3 has many partial solutions, but there is a full solution as well: PS_3 with the firing sequence $(t_4, t_4, t_2, t_0, t_1, t_3, t_3)$.

⁴Without the optimization, the algorithm would add the T-invariant $\{t_3, t_4\}$ to the solution vector again and again. In this particular case, this would lead to a full solution, but in general, adding the same invariant infinitely many times can lead to non termination.

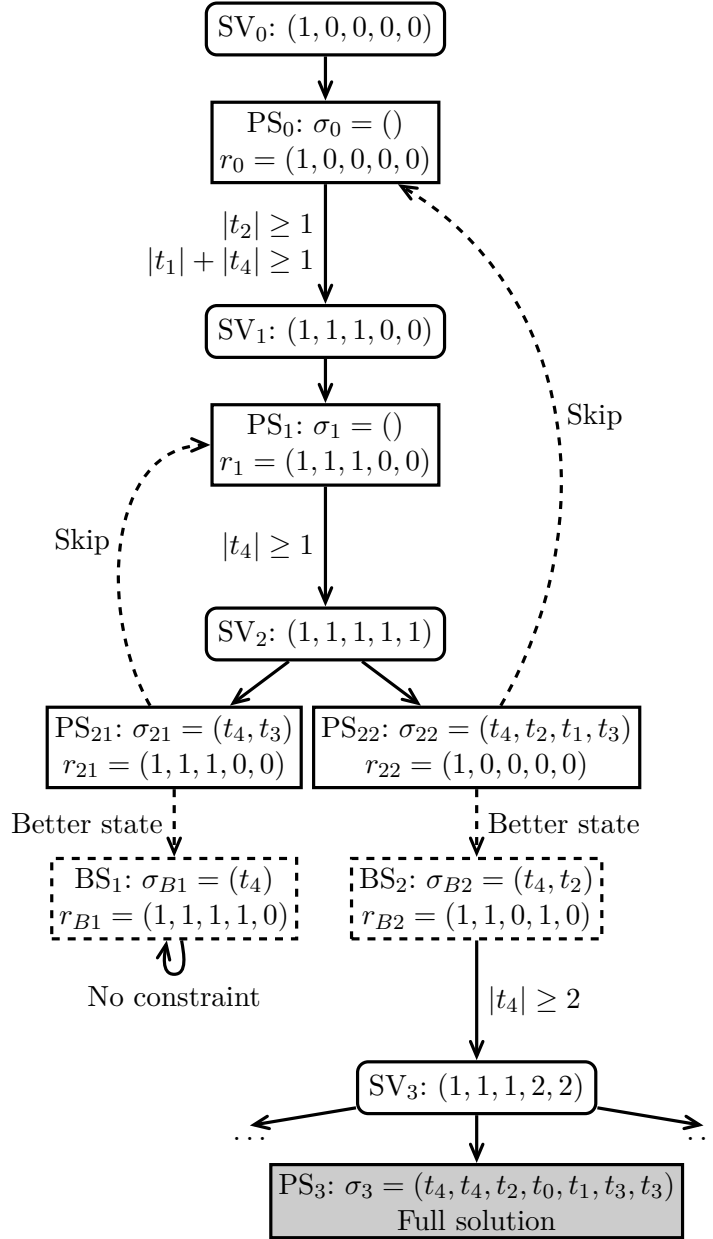


Figure 3.9: Solution space of the example seen in Figure 3.7

Chapter 4

Algorithmic contributions

In this chapter I present my work regarding the correctness (Section 4.2) and completeness (Section 4.3) of the algorithm. I also give a brief introduction to our previous findings [2, 3], which serve as a basis for my new contributions. We extended the algorithm to be able to solve new types of problems (Section 4.1) and we also had some results about the correctness and completeness of the algorithm. At the end of the chapter I present my formalization of the whole algorithm and the new contributions (Section 4.4).

4.1 Extensions of the algorithm

In our previous work [2, 3] we extended the algorithm to be able to handle two new types of problems: solving submarking coverability and handling Petri nets with inhibitor arcs.

4.1.1 Solving submarking coverability

In Section 2.2.2 I introduced the submarking coverability problem and its usefulness. In order to handle predicates of the form $Am \geq b$ in the CEGAR approach, we have to transform these conditions on places into conditions on transitions. This can be done by substituting m in the predicate with the state equation $m_0 + Cx = m$. This yields the inequality of the form

$$A(m_0 + Cx) \geq b,$$

which can be reordered in the form

$$(AC)x \geq b - Am_0.$$

This inequality can now be solved as an ILP problem for the firing count of transitions (x). The algorithm uses this inequality as the initial abstraction, and extends it with further (jump or increment) constraints.

4.1.2 Handling inhibitor arcs

The main problem with inhibitor arcs (Section 2.1.3) is that they do not appear in the incidence matrix (and thus, in the state equation) in any form. Therefore, a solution vector

of the state equation may not be realizable due to inhibitor arcs disabling some transitions. This means that tokens must be removed from some places. Our strategy in this case is to add such transitions to the solution that consume tokens from these places. We use increment constraints for this purpose, but generate them with a modified form of the three step algorithm (presented in Section 3.2.3):

- The first step is to build a dependency graph with transitions that could not fire due to inhibitor arcs and places disabling them. An arc from a place p to a transition t means that t is disabled by an inhibitor arc connecting to p , while the other direction indicates that firing t would consume tokens from p . Each source SCC of the graph is important, because tokens cannot be removed from them by other SCCs.
- The second step is to estimate the number of tokens to be removed. If T_i of the tuple (P_i, T_i, X_i) is not empty, any transition in T_i may enable all the others, so we find the transition that needs the least tokens to be removed. Otherwise, P_i contains only a single place, therefore the needs of the transitions in X_i must be fulfilled at once. This means that all the tokens have to be removed from the place of P_i .
- The third step is to construct a constraint for each SCC, by adding transitions to the solution vector that consume tokens from the places of the SCC.

4.2 Correctness of the algorithm

In our previous work [2, 3] we proved that the algorithm is incorrect due to the heuristic used for generating increment constraints, and we presented a method to detect such situations (Section 4.2.1). However, we were not able to provide a solution if the heuristic fails and in some special cases (using optimizations) we could not detect the problem. In this section I present my new contributions to detect the problem of the heuristic in every case (Section 4.2.3), and to be able to provide solutions even if the heuristic fails (Section 4.2.2).

4.2.1 Proof of the incorrectness

Although Theorem 1 states that a realizable solution can be reached by the CEGAR method, we found that the original algorithm [1] can give an incorrect answer in some special cases. The reason is that the heuristic used for generating increment constraints can over-estimate the number of required tokens. We proved this with the Petri net in Figure 4.1 and the reachability problem $(1, 0, 0, 0, 0, 0, 0, 2) \rightarrow (0, 1, 0, 0, 1, 0, 0, 2)$, i.e., a token is moved from p_0 to p_1 , and a token is produced in p_4 [2, 3].

We know that the vector $x_s = (1, 1, 1, 1, 1, 1, 1, 1)$ is a solution realizable by the firing sequence $\sigma_s = (t_1, t_2, t_0, t_5, t_6, t_3, t_7, t_4)$. However, the original algorithm does the following steps. At first, the minimal solution is $x = (1, 0, 1, 1, 1, 0, 0, 0)$, i.e., firing t_0, t_2, t_3, t_4 . Only t_0 is enabled from these transitions, so the only partial solution is $PS = (\emptyset, x, \sigma = (t_0), r = (0, 0, 1, 1, 1, 0, 0, 0))$. Then, the algorithm tries to find an increment constraint, for which the dependency graph can be seen in Figure 4.2.

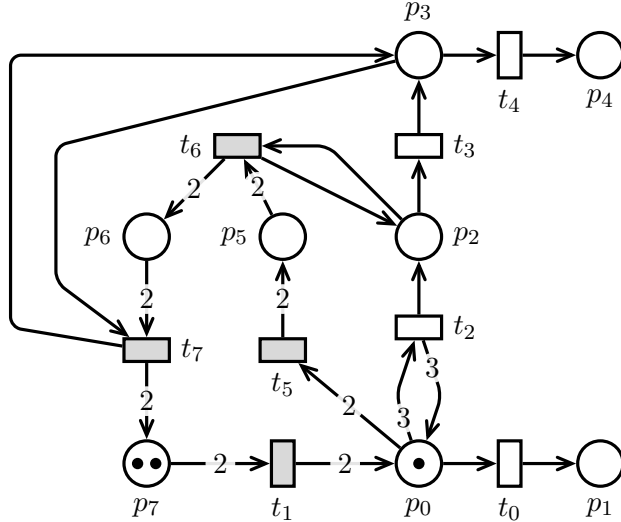


Figure 4.1: Proof of the incorrectness of the algorithm

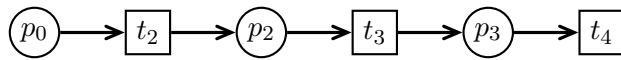


Figure 4.2: Dependency graph of the partial solution PS

The only source SCC is $SCC(\{p_0\}, \emptyset, \{t_2\})$, with zero tokens in p_0 (since firing t_0 consumed a token from there). The heuristic now estimates that three tokens are required in p_0 , where t_1 can produce tokens, so the increment constraint takes the form $2|t_1| \geq 3$. As a consequence, the T-invariant $\{t_1, t_5, t_6, t_7\}$ (gray transitions in Figure 4.1) is added to the solution vector twice. However, the previous invariant is constructed so that for each of its occurrence, a token has to be present in p_2 and p_3 in order to enable t_6 and t_7 . Note that these tokens can only be shifted towards p_4 , but in the target marking p_4 can hold only one token. Therefore, the problem can no longer be solved by the algorithm, so it gives the incorrect answer “not reachable”.

The problem here is that the heuristic over-estimated the tokens required in p_0 . Without firing t_0 , only two tokens would be missing from p_0 , which would mean that the T-invariant $\{t_1, t_5, t_6, t_7\}$ is added only once to the solution vector, resulting in the realizable solution x_s . This is a general problem of the original algorithm: it always tries to produce maximal firing sequences, although some transitions would not be practical to fire (t_0 in our example). Due to this, the estimated number of tokens in the final marking of the firing sequence can be higher than actually required.

Detecting over-estimation

In our improved algorithm [2, 3] we count the maximal number of tokens in each place during the firing sequence of a partial solution. If the final marking is not the maximal regarding any SCC, the heuristic might have over-estimated the number of tokens required¹. If such situation occurs and we do not find a full solution, we say that the problem cannot

¹For inhibitor SCCs, the minimal number of tokens is counted in each place. If the final marking is not the minimal regarding an inhibitor SCC, over-estimation may occur.

be decided.

The main goals of my thesis regarding the correctness of the algorithm are

- to find the solution even if the heuristic fails (Section 4.2.2)
- and to check if the previous method can always detect over-estimation (Section 4.2.3).

4.2.2 Providing a solution in case of over-estimation

I developed a new method that tries to find a solution in case of an over-estimation by the following way. Suppose that the heuristic estimated that n tokens are required in an SCC and I detected over-estimation. My first idea was to forget n , and estimate one instead. However, over-estimation is not a problem in most cases: the algorithm still finds a realizable solution, but not the minimal. Estimating one means a slow convergence to the actual number of missing tokens, so at first I always estimate n (even if I detect over-estimation). If no realizable solution can be found in that subtree, I backtrack and start a new search with $n = 1$.

Using this new method, the Petri net in Figure 4.1 can now be solved with the reachability problem $(1, 0, 0, 0, 0, 0, 2) \rightarrow (0, 1, 0, 0, 1, 0, 2)$. Figure 4.3 shows the relevant part of the solution space of this problem, illustrating how the new method works.

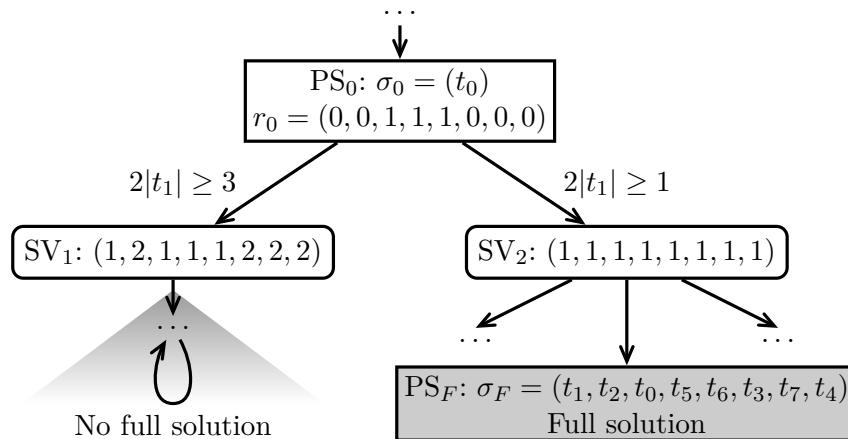


Figure 4.3: Part of the solution space showing how solutions can be found in case of over-estimation

The partial solution where the over-estimation occurs is labeled by PS_0 . At first I try the original estimation ($n = 3$), yielding a new solution vector $SV_1 : (1, 2, 1, 1, 1, 2, 2, 2)$. Since there are no full solutions in this subtree, I backtrack to PS_0 and now estimate $n = 1$. This way, the realizable solution vector $SV_2 : (1, 1, 1, 1, 1, 1, 1, 1)$ is found, with a full solution PS_F .

Limitations, future work

During my work I found that the heuristic may not only over-estimate the required tokens, it may also determine wrong places to put tokens in. As an example, consider the Petri net in Figure 4.4 with the reachability problem $(0, 1, 0, 1) \rightarrow (1, 0, 1, 1)$, i.e., move the token

from p_1 to p_2 , and produce a token in p_0 . A possible solution is the vector $x_s = (1, 1, 1, 1)$, realized by the firing sequence $\sigma_s = (t_3, t_0, t_1, t_2)$.

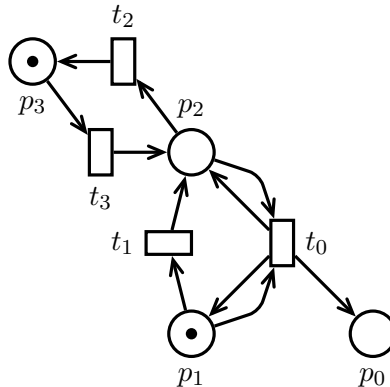


Figure 4.4: *In some special cases, even estimating $n = 1$ does not help finding a solution.*

The algorithm does the following steps. The minimal solution is $x_0 = (1, 1, 0, 0)$, i.e., firing t_0 and t_1 . Only t_1 is enabled, thus one partial solution $PS_0 = (\emptyset, x_0, \sigma_0 = (t_1), r_0 = (1, 0, 0, 0))$ can be found. The marking reached by σ_0 is $(0, 0, 1, 1)$, where $n = 1$ token is missing from p_1 to enable t_0 . None of the transitions can produce tokens in p_1 , so the algorithm cannot find any constraint. Before firing t_1 , p_1 had one token, so the algorithm detects over-estimation. However, a new search cannot be started, since the original estimation is also $n = 1$.

Without firing t_1 , a token would be missing from p_2 , where the T-invariant $\{t_2, t_3\}$ could help. The problem is that the heuristic tries to produce tokens in a place (p_1), which lacks tokens in the final marking, but had the required number of tokens at some point of the firing sequence (σ_0). Finding a solution in such situations is an aim of my future work.

4.2.3 Detecting over-estimation structurally

The over-estimation detecting method presented before is correct as long as every intermediate marking of a partial solution is observed. However, using the subtree omission optimization (Section 3.2.5), firing sequences only different in the order of transitions are skipped, thus some intermediate markings are lost.

As an example, consider the Petri net in Figure 4.5 with the submarking coverability problem to reach a marking m with $m(p_1) = 1$, $m(p_3) = 1$.

The minimal solution is $x_0 = (1, 0, 1)$, i.e., firing t_0 and t_2 . Since only t_0 is enabled, the only partial solution is $PS_0 = (\emptyset, x_0, \sigma_0 = (t_0), r = (0, 0, 1))$. The algorithm finds that t_2 is disabled by p_0 , where three tokens are required (t_0 consumed one token from p_0). This is an over-estimation, because p_0 had more tokens at the initial marking. The constraint $|t_1| \geq 3$ makes the state equation infeasible, but a new search is started with a constraint $c_1 = |t_1| \geq 1$, leading to a new solution $x_1 = (1, 1, 1)$. There are two partial solutions now, where t_0 and t_1 fire in a different order. In both cases, t_2 is disabled by p_0 with two tokens missing at the final marking (t_0 consumes, but t_1 produces a token in p_0). This

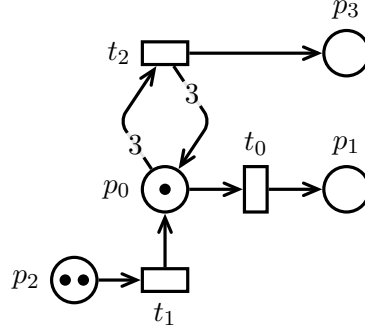


Figure 4.5: A net, where over-estimation cannot be detected if subtree omission is used

would imply that t_1 is added twice to the solution vector, but p_2 has only one token left, so no solution would be found. When the algorithm tries to detect over-estimation, the following happens for the two partial solutions:

- $PS_{11} = (\{c_1\}, x_1, \sigma_{11} = (t_0, t_1), r_{11} = (0, 0, 1))$: p_0 has one token at the initial and final marking, and zero tokens after firing t_0 . The over-estimation detection method fails, since the final marking of p_0 is not less than any other marking.
- $PS_{12} = (\{c_1\}, x_1, \sigma_{12} = (t_1, t_0), r_{12} = (0, 0, 1))$: p_0 has one token at the initial and final marking, and two tokens after firing t_1 . This is an over-estimation, so t_1 is added only once, yielding a realizable solution $x_s = (1, 2, 1)$.

Note that PS_{11} and PS_{12} are only different in the order of transitions. Using the subtree omission, we do not know, which of them is skipped. If PS_{12} is skipped, the algorithm gives an incorrect answer.

To overcome this problem, I developed a method that can detect over-estimation independent from the order of transitions, using only structural information from the Petri net. The maximal possible token count of a place p in a partial solution $PS = (\mathcal{C}, x, \sigma, r)$ is determined by the following formula²:

$$m_{max}(p) = m_0(p) + \sum_{t \in T, C(p,t) > 0} (\wp(\sigma)(t)) \cdot C(p, t).$$

The formula assumes an ideal case, where only transitions t with $C(p, t) > 0$ (i.e., that produce tokens in p) fire from σ . The product $(\wp(\sigma)(t)) \cdot C(p, t)$ represents the total number of tokens produced by t in p in the firing sequence σ . Assuming an ideal case can sometimes lead to false detections. This may yield some computational overhead, but the result of the algorithm is correct.

Using this new approach, the algorithm can solve the previous example. Since this method is independent from the order of transitions, it can detect the over-estimation both at PS_{11} and PS_{12} .

²For inhibitor SCCs, transitions t with $C(p, t) < 0$ should be counted.

4.3 Completeness of the algorithm

In our previous work [2, 3] we found several problems that the original algorithm [1] could not solve. We developed some extensions to overcome these problems, but we proved that even the improved algorithm [2, 3] is incomplete. In this section I briefly introduce our previous work (Section 4.3.1) and present my new extension to the iteration strategy of the algorithm (Section 4.3.2). I also improve the T-invariant filtering optimization with an extra criterion (Section 4.3.3).

4.3.1 Previous work

This section briefly introduces our previous results [2, 3] regarding the completeness of the algorithm.

Total ordering of the intermediate markings

When a partial solution $PS = (\mathcal{C}, x, \sigma, r)$ is skipped by the T-invariant filtering optimization, the algorithm checks if it was closer to enabling a transition t (with $r(t) > 0$) during the firing of σ . The original algorithm does this by “counting the minimal number of missing tokens for firing t in the intermediate markings occurring” [1].

In our previous work we found that this criterion is not general enough: in some cases the total number of missing tokens may not be less, but different places are lacking tokens, where additional ones could be produced. In our improved algorithm, an intermediate marking m_i is considered better than the final marking m' of the firing sequence σ , if there is a transition $t \in T$ with $r(t) > 0$ and a place $p \in P$ with $(p, t) \in E$, for which $m'(p) < w^-(p, t) \wedge m_i(p) > m'(p)$ holds [3]. This means that t is disabled by p at the final marking m' and less tokens are missing to enable t in the intermediate marking m_i . Figure 3 of [3] is an example where the previous definition helps to extend the set of decidable problems.

T-invariant filtering and subtree omission

As I already stated in Section 4.2.3, intermediate markings can be lost using the subtree omission optimization. This is not only a problem for over-estimation detection, but also for finding better intermediate markings in case of a skipped partial solution. In such situations we rebuild the partial solution tree without subtree omission. Note that this was not an acceptable option for detecting over-estimation, since it must be done for every partial solution. However, it is a reasonable option if a partial solution gets skipped by the T-invariant filtering optimization. Figure 4 of [3] is an example where the order of transitions matters for finding better markings. This approach yields a computational overhead in some cases, but full solutions can be lost otherwise.

New termination criterion

We also developed a new criterion for termination, which is proved to keep all full solutions [2, 3]. This new criterion is applied during the increment constraint generating process. Suppose that the algorithm found a set of places $P' \subseteq P$ with the estimated number of required tokens n (for some partial solution PS). Before trying to generate a constraint, we can check if the following inequalities hold for some marking m' :

$$\sum_{p_i \in P'} m'(p_i) \geq n$$

$$\forall p_j \in P : m'(p_j) \geq 0.$$

The first inequality ensures that at least n tokens are produced in the places of P' and the others guarantee that each place has non-negative number of tokens under m' . These inequalities define a submarking coverability problem, which can be solved easily by the ILP solver. If there is no solution, the necessary criterion for reachability does not hold, thus n tokens cannot be produced in the places of P' . In this case PS can be skipped without losing any full solution. This new approach can cut the search space efficiently and it can prevent non-termination. It also extends the set of decidable problems, especially where the target marking is not reachable. Figure 5 of [3] presents such an example.

Proof of the incompleteness

Despite the previous extensions, we proved that the algorithm is still incomplete due to its iteration strategy. A proof can be found in [2] and [3], but I present a new and simpler proof on a smaller Petri net here. Consider the Petri net in Figure 4.6 with the reachability problem $(0, 1, 0) \rightarrow (1, 1, 0)$, i.e., I want to produce a token in p_0 . The vector $x_s = (1, 1, 1, 1, 1)$ is a solution, realized by the firing sequence $\sigma_s = (t_3, t_1, t_0, t_2, t_4)$.

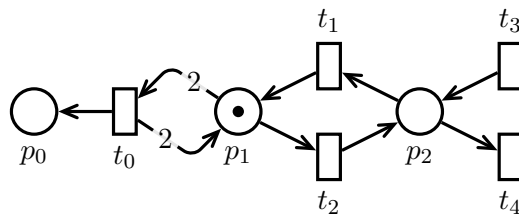


Figure 4.6: A counterexample of completeness

The algorithm does the following steps. The minimal solution vector is $x_0 = (1, 0, 0, 0, 0)$, i.e., firing t_0 . Since t_0 is not enabled, the only partial solution is $PS_0 = (\emptyset, x_0, \sigma_0 = (), r_0 = (1, 0, 0, 0, 0))$. The algorithm finds that an additional token is required in p_1 and only t_1 can satisfy this need. With an increment constraint $c_1 : |t_1| \geq 1$, the T-invariant $\{t_1, t_2\}$ is added to the new solution vector $x_1 = (1, 1, 1, 0, 0)$. Only t_2 and t_1 can fire (in this order), thus the only partial solution for x_1 is $PS_1 = (\{c_1\}, x_1, \sigma_1 = (t_2, t_1), r_1 = r_0)$.

This partial solution is skipped by the T-invariant filtering optimization, since the difference of σ_0 and σ_1 is a T-invariant $(\{t_1, t_2\})$ and $r_1 = r_0$. Furthermore, there are no

better intermediate states, since no additional token was “borrowed” from the T-invariant $\{t_1, t_2\}$. The algorithm terminates at this point, leaving the problem undecided.

The problem is that the algorithm does not recognize that although $\{t_1, t_2\}$ can fire, it only circulates the same token, instead of “lending” a new one. An extra token could be produced in p_2 (and then moved in p_1) using the T-invariant $\{t_3, t_4\}$. However, $\{t_3, t_4\}$ is not connected directly to p_1 (where the tokens are missing), so the iteration strategy of the algorithm does not try to involve it. In the following subsection I propose an extension to the iteration strategy in order to involve such “distant” invariants into the solution vector.

4.3.2 Involving distant T-invariants

Definition 1 (Distant invariant) *A T-invariant x is a distant invariant for a set of places P_i of an SCC if there are no edges between the transitions of x and the places of P_i , but they are connected indirectly through other places and transitions.*

When a partial solution is skipped due to a T-invariant, it means that this invariant was fired, but it could not “lend” enough tokens. The basic idea of involving distant T-invariants is quite simple: try to produce tokens in any place connected to the filtered T-invariant. If some tokens can be produced, the filtered invariant will then be able to transfer them indirectly to the place that lacks tokens. However, there are two problems to be solved:

- How many tokens should be produced in the places of the filtered invariant?
- When should this process terminate? If the distant invariant cannot help, adding it again will lead to non-termination.

Number of tokens produced in the invariant

There are two possibilities for the number of tokens produced in a T-invariant:

- Produce one token at a time, and repeat this process if it was not enough.
- Estimate the required number of tokens.

Estimating the required number of tokens is a hard problem, since the sum of the tokens in the places of an invariant may change during firing. As an example, consider the T-invariant $\{t_1, t_2\}$ in Figure 4.7. One token in p_2 yields two tokens in p_1 . Over-estimation can also be a problem: the final marking of the T-invariant may not be the “best” state regarding the number of tokens. Therefore, I adapt the former strategy, namely producing one token at a time.

Termination criterion

Suppose that a partial solution PS was skipped and the algorithm tried to produce tokens using distant invariants. If the distant invariant can also not help, adding it again can

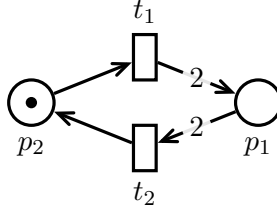


Figure 4.7: *T-invariant example*

lead to non-termination. I developed a new algorithm (Algorithm 1) to decide if a distant T-invariant should be added or not.

The input of the algorithm is a partial solution PS' that was skipped due to another partial solution PS . The number of better intermediate states is also an input, since it indicates whether the distant invariant helped. Each partial solution stores a list of T-invariants. This list keeps track of the invariants, in which the partial solution and its ancestors tried to put tokens.

At first, I calculate the difference between the firing sequences of PS and PS' (ActTinv), i.e., the invariant that actually caused filtering PS' . Then, I initialize the actual list of invariants to put tokens in (ActDistantInvariants) with an empty list. At this point, there are two possible cases:

- If PS was also skipped that means some distant invariants were already involved. If these invariants helped getting closer to enable a transition in the remainder, then there are better intermediate markings ($bs > 0$). In this case these invariants should be added again, so the actual list of invariants (and also the list of PS') is the same as the list of PS .
- Otherwise, (if PS was not skipped or $bs = 0$) I try to involve a new distant invariant. At first I get the latest invariant that PS tried to help (or an empty set, if PS was not skipped). If the actually filtered invariant is not a subset of the latest invariant of PS , I take their union and put them in the actual list of invariants. However, for PS' I store both the previous invariants and the actual (the union).

These conditions ensure that a distant invariant is only added as long as it has positive effect ($bs > 0$) and when a new distant invariant is involved, it must be greater than the previous (union with non-subset).

If the actual list of invariants is not an empty list, I take each invariant and get the places connected to the transitions of the invariant. Creating a constraint for these places is done by the third step of the increment constraint generating heuristic. If no constraint can be found (i.e., no transitions can produce tokens in an invariant), the algorithm returns no new solution vector. Otherwise the new constraints (c_i) are added to the constraints of PS' and I solve the ILP problem for a new solution vector.

The example in Figure 4.6 can now be trivially solved: PS_1 is filtered due to PS_0 and the T-invariant $\{t_1, t_2\}$. Since PS_0 was not filtered, the algorithm produces tokens in places connected to $\{t_1, t_2\}$. These are places p_1 and p_2 , where only t_3 can increase the

token count. Thus, the distant T-invariant $\{t_3, t_4\}$ is added and the realizable solution $x_s = (1, 1, 1, 1, 1)$ is found.

Algorithm 1: Distant invariant algorithm

Input : PS' : Partial solution skipped
 PS : Partial solution that caused skipping PS'
 bs : Number of better intermediate markings for PS'

Output : x : New solution vector found by involving distant invariants

- 1 ActTinv \leftarrow difference invariant between PS and PS' ;
- 2 ActDistantInvariants $\leftarrow \emptyset$;
- 3 **if** PS was already skipped $\wedge bs > 0$ **then**
- 4 | ActDistantInvariants \leftarrow Distant invariants of PS ;
- 5 | Distant invariants of PS' \leftarrow ActDistantInvariants;
- 6 **end**
- 7 **else if** ActTinv \notin Latest distant invariant of PS **then**
- 8 | ActDistantInvariants \leftarrow {Latest distant invariant of $PS \cup$ ActTinv};
- 9 | Distant invariants of PS' \leftarrow Distant invariants of $PS \cup$ {ActDistantInvariants};
- 10 **end**
- 11 **if** ActDistantInvariants $\neq \emptyset$ **then**
- 12 | ConstrList \leftarrow constraints of PS' ;
- 13 | **for each** T-Invariant $T_i \in$ ActDistantInvariants **do**
- 14 | | $c_i \leftarrow$ constraint to produce one token in places connected to T_i ;
- 15 | | **if** no constraint can be found **then** no new solution vector is returned;
- 16 | | ConstrList \leftarrow Constrlist $\cup \{c_i\}$;
- 17 | **end**
- 18 | $x \leftarrow$ solve the state equation with ConstrList;
- 19 **end**

A complex example

I present here a complex example, showing all aspects of the new algorithm. Consider the Petri net in Figure 4.8 with the reachability problem $(0, 1, 0, 0, 2) \rightarrow (1, 1, 0, 0, 2)$, i.e., producing a token in p_0 .

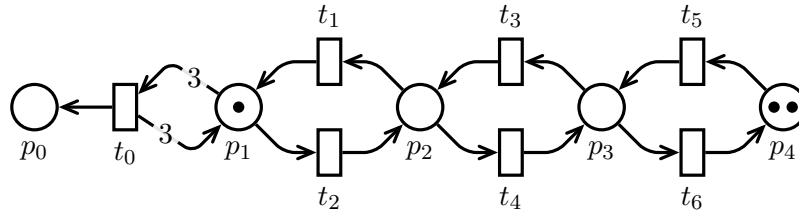


Figure 4.8: A complex example for involving distant invariants

The relevant part of the solution space can be seen in Figure 4.9. The minimal solution is $x_0 = (1, 0, 0, 0, 0, 0, 0)$, i.e., firing t_0 , with a single partial solution $PS_0 = (\emptyset, x_0, \sigma_0 = (), r_0 = x_0)$. The algorithm finds that two tokens are missing from p_1 , thus the invariant $\{t_1, t_2\}$ is added twice by the constraint $c_1 : |t_1| \geq 2$. The new solution vector is $x_1 = (1, 2, 2, 0, 0, 0, 0)$ with one partial solution $PS_1 = (\{c_1\}, x_1, \sigma_1 = (t_2, t_1, t_2, t_1), r_1 = r_0)$.

The invariant $\{t_1, t_2\}$ could fire, but did not enable t_0 so PS_1 is skipped due to PS_0 . Also, there are no better intermediate markings, since $\{t_1, t_2\}$ did not produce any extra tokens.

At this point, the new algorithm tries to produce a token in p_1 and p_2 by distant invariants. This implies that the T-invariant $\{t_3, t_4\}$ is added once to the new solution $x_2 = (1, 2, 2, 1, 1, 0, 0)$ by the constraint $c_2 : |t_3| \geq 1$. There are several partial solutions for x_2 , since $\{t_3, t_4\}$ can be fired or not, and the order of transitions can also be different, which is irrelevant now.

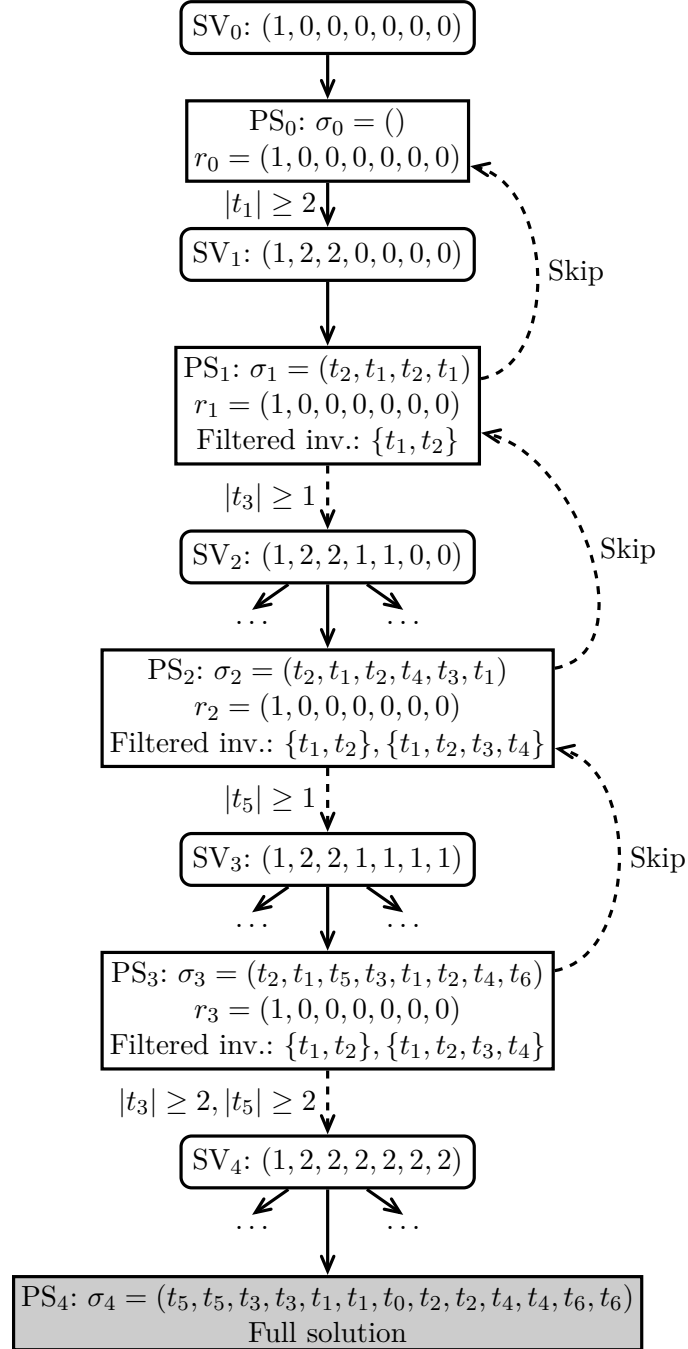


Figure 4.9: Solution space of the example on distant invariants

Suppose that we proceed the partial solution $PS_2 = (\{c_1, c_2\}, x_2, \sigma_2 = (t_2, t_1, t_2, t_4, t_3, t_1), r_2 = r_1)$ where the distant invariant $\{t_3, t_4\}$ could fire but did not enable t_0 . Thus,

PS_2 is skipped due to PS_1 . PS_1 was already skipped and there are no better intermediate markings ($\{t_3, t_4\}$ did not “lend” any extra tokens). However, $\{t_3, t_4\} \not\subseteq \{t_1, t_2\}$ so now the algorithm tries to produce a token in $\{t_1, t_2\} \cup \{t_3, t_4\}$, i.e., places p_1, p_2, p_3 . In the new solution vector $x_3 = (1, 2, 2, 1, 1, 1, 1)$ another distant invariant $\{t_5, t_6\}$ is added by the constraint $c_3 : |t_5| \geq 1$.

There are several partial solutions as well, but suppose that we proceed to $PS_3 = (\{c_1, \dots, c_3\}, x_3, \sigma_3 = (t_2, t_1, t_5, t_3, t_1, t_2, t_4, t_6), r_3 = r_2)$. The distant invariant $\{t_5, t_6\}$ fired, but it did not enable t_0 . Therefore, PS_3 is skipped due to PS_2 . Although PS_2 was already skipped, involving $\{t_5, t_6\}$ resulted in an extra token, so there are better intermediate states. The algorithm therefore, tries to produce tokens in the invariants of PS_2 , which are $\{t_1, t_2\}$ and $\{t_1, t_2, t_3, t_4\}$. The constraints $c_4 : |t_3| \geq 2$ and $c_5 : |t_5| \geq 2$ yield a new solution $x_4 = (1, 2, 2, 2, 2, 2, 2)$, which can be realized by the full solution $PS_4 = (\{c_1, \dots, c_5\}, x_4, \sigma_4 = (t_5, t_5, t_3, t_3, t_1, t_1, t_0, t_2, t_2, t_4, t_4, t_6, t_6), 0)$.

Limitations, future work

This new approach also has some limitations. If the T-invariants have edge weights greater than one, producing one token at a time may not help. As an example, consider the Petri net in Figure 4.10 with the reachability problem $(0, 1, 0) \rightarrow (1, 1, 0)$. After $\{t_1, t_2\}$ is added and filtered, the new approach involves $\{t_3, t_4\}$ once, which yields one token in p_2 . However, t_1 requires two tokens to fire, so this token cannot be moved in p_1 . There are no better intermediate markings, since we only consider enabling t_0 .

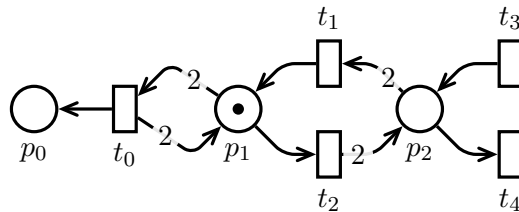


Figure 4.10: *Limitations of the new approach on distant invariants*

There are two possible solutions to overcome this problem:

- Count the arc weights and estimate the required number of tokens instead of producing one token at a time.
- Better intermediate markings can be considered not only for the disabled transition, but for the invariant as well.

My future plan is to investigate these methods in order to extend the new approach.

4.3.3 Extending the filtering optimization

The T-invariant filtering optimization is an important part of the algorithm, since it can prevent non-termination. However, I found some cases, where the definition is not general enough and therefore, it cannot detect infinite loops of T-invariants.

Remainder based filtering

There are special cases, where T-invariants may either fire or not. As an example consider the Petri net in Figure 2.4 and suppose that each transition must fire once. This is realizable with the firing sequence $\sigma_0 = (t_0, t_2, t_3, t_1)$. However, $\sigma_1 = (t_0, t_1)$ is also a maximal firing sequence for this problem, since t_2 and t_3 are disabled after σ_1 . In such cases both partial solutions are found and processed. To differentiate between the two cases, I say that the invariant is added to the firing sequence or to the remainder. The first case can be detected by the T-invariant filtering optimization. However, I found that the second case can also lead to non-termination if there are at least two T-invariants (T_1, T_2) with this property. At first, T_1 is added without firing. Then T_2 is also added without firing, but now T_1 is fired. Finally T_1 is added again without firing, but now T_2 fires and this process is repeated. The T-invariant filtering optimization cannot detect this, since the remainder vector also changes in every step.

To overcome this problem, I detect when a T-invariant is added to the remainder, i.e., a partial solution $PS = (\mathcal{C}, x, \sigma, r)$ was found between the ancestors of $PS' = (\mathcal{C}', x + y, \sigma, r + y)$. However, PS' cannot be filtered immediately, since the remainder is different and the abstraction refinement may add new invariants that can help. I found that in most of the cases where PS' yielded non-termination, PS was a partial solution that could also be filtered by the original criterion for filtering. Therefore, I only skip the partial solution PS' based on the remainder vector, if PS can also be skipped based on the firing sequence.

Breadth first search between the SCCs

The extended filtering criterion can now detect if T-invariants are added either to the firing sequence or to the remainder without helping a transition in the remainder. However, when there are more than one SCCs, multiple T-invariants can be added in one step. If one of them is added to the firing sequence and the other is added to the remainder it cannot be detected, since both the firing sequence and the remainder changes. Therefore, I only deal with one SCC at a time. If an SCC does not lead to a full solution, I backtrack to the others.

4.4 Pseudo code

In this section I present my formalization of the algorithm and the new contributions with pseudo codes. The algorithm is split into several parts in order to be more understandable. Sub-algorithms are denoted by underlining in the pseudo code.

Algorithm 2 shows the main iteration loop of the CEGAR approach. The solution space is traversed using depth first search with a stack for backtracking purposes. The partial solution storing optimization is implemented with a set of partial solutions called “PScatalog”. The stack and the catalog are both accessible for the sub-algorithms.

At first, the state equation is solved without constraints and the solution is pushed in the stack. While the stack is not empty I take the top element, which is either a partial

solution or a solution vector. In both cases this element is processed by sub-algorithms (Algorithm 3 and 4). These sub-algorithms put new solution vectors or partial solutions in the stack (if they find any). The loop continues until a full solution is found or there is no possibility to backtrack.

Algorithm 3 shows how a partial solution is processed. At first I check whether it can be skipped by the T-invariant filtering optimization (Algorithm 5):

- If the partial solution is skipped, I search for better intermediate states (Algorithm 6) and try to involve distant invariants (Algorithm 1).
- Otherwise, I try to obtain new solution vectors by increment constraints (Algorithm 7).

Algorithm 2: Main loop of the CEGAR algorithm

Input : PN or PN_I : Petri net
 RP : Parameters of the reachability (or submarking coverability) problem

Output : Result: “Reachable” | “Not reachable” | “Not decidable”

- 1 Stack: Stack of solution vectors and partial solutions ;
- 2 PScatalog: Set of all partial solutions that occurred ;
- 3 Stack $\leftarrow \emptyset$;
- 4 PScatalog $\leftarrow \emptyset$;
- 5 **if** *The state equation has a solution x_0* **then** Stack $\leftarrow x_0$;
- 6 **while** Stack $\neq \emptyset$ **do**
- 7 $S \leftarrow$ Pop an item from the stack ;
- 8 **if** S is a partial solution **then** Process partial solution S ;
- 9 **else if** S is a solution vector **then**
- 10 Process solution vector S ;
- 11 **if** A full solution is found for S **then** Result \leftarrow “Reachable”;
- 12 **end**
- 13 **end**
- 14 **if** A partial solution was skipped or over-estimation occurred **then**
- 15 Result \leftarrow “Not decidable”;
- 16 **end**
- 17 **else** Result \leftarrow “Not reachable”;

Algorithm 3: Partial solution processing algorithm

Input : PS : Partial solution to be processed

- 1 **if** PS can be skipped due to a partial solution PS_0 **then**
- 2 Stack, PScatalog \leftarrow Better intermediate states for PS ;
- 3 Stack \leftarrow Solution vectors by involving distant invariants for PS ;
- 4 **end**
- 5 **else**
- 6 Stack \leftarrow Solution vectors with increment constraints for PS ;
- 7 **end**

Algorithm 4 presents how solution vectors are processed. At first, I try to find pairwise incomparable solutions by jump constraints. Then, I build the tree of partial solutions

with the optimization methods. If a full solution is found, it is returned, otherwise all the partial solutions are pushed in the stack for further processing.

Algorithm 5 presents the formalization of the T-invariant filtering optimization. I loop through each partial solution PS_E of each ancestor solution vector of PS . At first I check whether PS can be skipped based on the firing sequence. If not, the remainder vector is also checked.

Algorithm 6 finds better intermediate states for a skipped partial solution PS . I build the tree of partial solutions again, but this time without optimizations. Since I only want firing sequences different in the order of transitions, I work with $\wp(\sigma)$ instead of x . Then, for each intermediate marking I check the criteria presented in Section 4.3.1. I also check whether this intermediate marking has already been found.

Algorithm 7 is used to generate increment constraints for partial solutions that were not skipped. It is the three step algorithm presented in Section 3.2.3 extended with the new contributions. At first I build the dependency graph and find the source SCCs. Then, for each SCC I estimate the number of tokens and construct a constraint. If over-estimation is detected, another constraint is generated with $n = 1$. Then I add the constraints to the previous ones and solve the ILP problem. This way, the number of new solution vectors is between zero and twice the number of SCCs.

Algorithm 4: Solution vector processing algorithm

Input : x : Solution vector
Output : FS : Full solution (if found)
1 Stack \leftarrow Solution vectors with jumps;
2 Build the tree of partial solutions for x using stubborn sets and subtree omission;
3 **if** A full solution PS_{FULL} is found **then** $FS = PS_{FULL}$;
4 **else** Stack, PScatalog \leftarrow Partial solutions for x ;

Algorithm 5: Checking if a partial solution can be skipped

Input : $PS(\mathcal{C}, x, \sigma, r)$: Partial solution
Output : PS_0 : Partial solution that caused filtering PS (if such PS_0 exists)
1 **for** Each solution vector SV from the ancestors of PS **do**
2 **for** Each partial solution $PS_E(\mathcal{C}_E, x_E, \sigma_E, r_E)$ of SV **do**
3 **if** $\wp(\sigma) - \wp(\sigma_E)$ is a T-invariant and $r = r_E$ **then**
4 PS can be skipped;
5 $PS_0 = PS_E$;
6 **end**
7 **else if** $\wp(\sigma) = \wp(\sigma_E)$ and $r - r_E$ is a T-invariant **then**
8 **if** PS_E can be skipped based on the firing sequence **then**
9 PS can be skipped;
10 $PS_0 = PS_E$;
11 **end**
12 **end**
13 **end**
14 **end**

Algorithm 6: Finding better intermediate states

Input : $PS(\mathcal{C}, x, \sigma, r)$: Partial solution
Output : BSs : Set of better intermediate states

- 1 $m' \leftarrow$ marking reached by firing σ ;
- 2 Build the partial solution tree for $\wp(\sigma)$ without optimizations;
- 3 **for** *Each intermediate marking* m_i **do**
- 4 **if** $\exists t, p$ with $r(t) > 0$ and $m'(p) < w^-(p, t) \wedge m_i(p) > m'(p)$ **then**
- 5 m_i is a better state;
- 6 **if** m_i was not yet found **then** $BSs \leftarrow m_i$;
- 7 **end**
- 8 **end**

Algorithm 7: Finding increment constraints

Input : $PS(\mathcal{C}, x, \sigma, r)$: Partial solution
Output : SVs : New solution vectors (if found)

- 1 Build the dependency graph;
- 2 Find source SCCs (normal and inhibitor as well);
- 3 **for** *Each SCC* **do**
- 4 $n \leftarrow$ estimated number of tokens required for this SCC;
- 5 **if** *The necessary criterion holds* **then**
- 6 $c \leftarrow$ constraint to produce/remove n tokens from this SCC;
- 7 $SVs \leftarrow$ solve the state equation with $\mathcal{C} \cup \{c\}$;
- 8 **end**
- 9 **if** *Over-estimation is detected and* $n \neq 1$ **then**
- 10 $c' \leftarrow$ constraint to produce/remove 1 token from this SCC;
- 11 $SVs \leftarrow$ solve the state equation with $\mathcal{C} \cup \{c'\}$;
- 12 **end**
- 13 **end**

Chapter 5

Implementation

In this chapter I present my implementation of the CEGAR algorithm including my new contributions. At first I present the framework used for the development (Section 5.1) and the architecture (Section 5.2) of the components. Then, I give a brief introduction on the functionality of the tool (Section 5.3). At the end of this chapter, the details of the implementation are presented (Section 5.4).

5.1 The PetriDotNet framework

I implemented the algorithm as a plug-in for the *PetriDotNet* framework¹ [15], which is an application for editing, simulating and analyzing Petri nets. Figure 5.1 shows the main screen of the framework. It is written in C# programming language, and it can be extended with plug-ins easily through its public interface.

Public interface

Each plug-in must contain a class implementing the interface `IPDNPlugin`. This class gets a reference of the application (`PDNAppDescriptor`). The actual Petri net of the editor can be obtained by the `CurrentPetriNet` property. The algorithm creates its own representation of the net (Section 5.4.2) using the following methods of the `PetriNet` class:

- The method `GetNonVisualPlaces()` returns a list of `Places`. The algorithm stores the names of the places and assigns a unique ID to each of them.
- The method `GetTransitions()` returns a list of `Transitions`. The algorithm stores the transitions similar to the places.
- The method `GetNonVisualEdges()` returns a list of `Edges`. Each edge has a `Source` and `Target` property, which is either a place or a transition. Edges also have an integer `Weight` and a boolean `Inhibitor` property. The algorithm stores the edges in matrices.

¹*PetriDotNet* Version 1.3.4804.24282

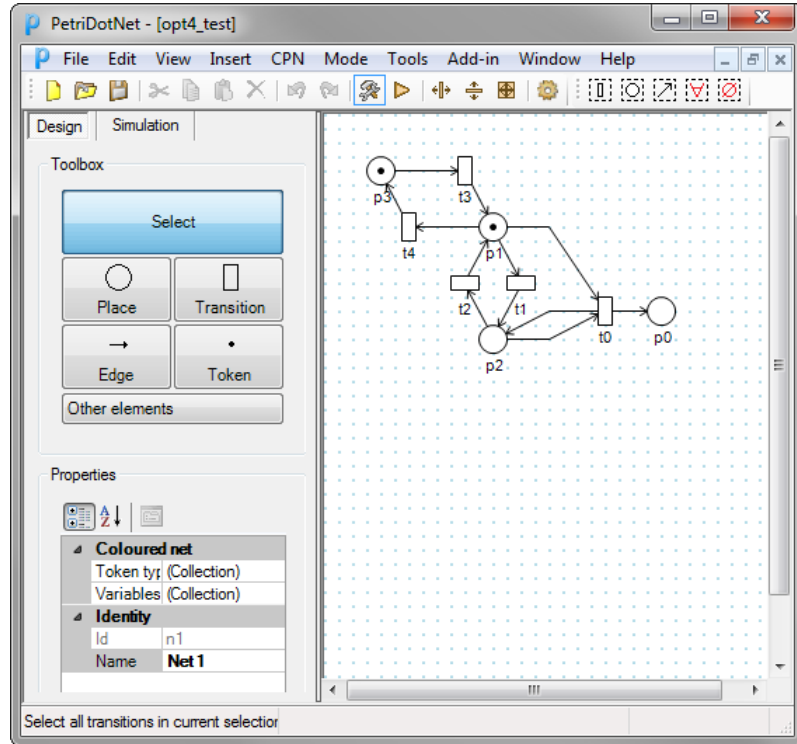


Figure 5.1: Main screen of *PetriDotNet*

5.2 Architecture

An overview of the architecture can be seen in Figure 5.2. The Petri net can be created and edited graphically in the *PetriDotNet* framework. When the CEGAR plug-in is started, it gets a reference of the current Petri net. After setting the parameters of the reachability (or submarking coverability) problem, the plug-in starts to explore the solution space using an ILP solver. For this purpose I used the *lpsolve* tool [16], which is presented in Section 5.4.1. The abstraction is refined in an iterative process, so the CEGAR plug-in may call the solver several times. If a full solution is found, the firing sequence can be simulated graphically in *PetriDotNet*.

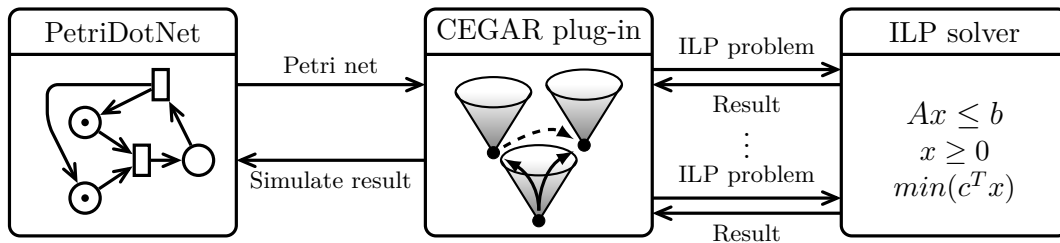


Figure 5.2: Overview of the architecture

5.3 Functionality

This section gives a brief introduction on installing and using the plug-in. It was an important aspect to create a convenient tool that can be used easily without deep knowledge

about the algorithm.

5.3.1 Deployment

The plug-in consist of three files and three directories (see Figure 5.3). These files and directories should be copied in the same structure under the “add-in” folder of *PetriDotNet*.

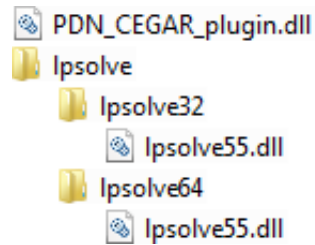


Figure 5.3: Files and directories of the plug-in

5.3.2 Overview of the GUI

The plug-in can be started from *PetriDotNet* with the “Reachability (CEGAR)” item of the “Add-in” menu. The main window of the plug-in can be seen in Figure 5.4.

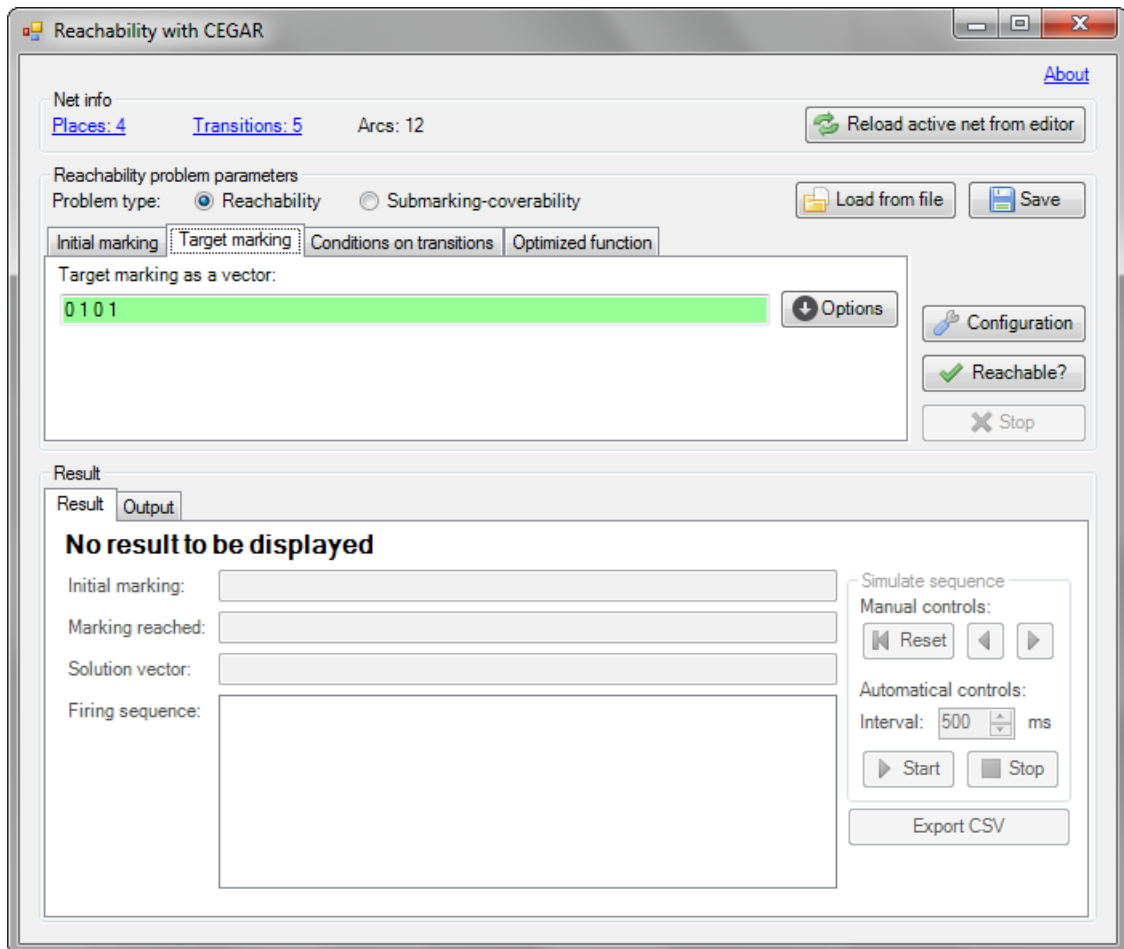


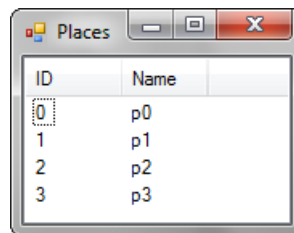
Figure 5.4: Main window of the CEGAR plug-in

The window is divided into the following three sections:

- information about the net,
- parameters of the reachability problem,
- result of the analysis.

5.3.3 Information about the net

The top section displays information about the currently loaded Petri net. If the net is changed in the editor, it must be reloaded manually with the “Reload active net from editor” button. The number of places, transitions and edges is also displayed. If the “Places” or “Transitions” label is clicked, the plug-in displays the ID and name of each place or transition in a pop-up dialog (Figure 5.5).



ID	Name
0	p0
1	p1
2	p2
3	p3

Figure 5.5: Place names with IDs

5.3.4 Parameters of the reachability problem

The type of the problem (reachability or submarking coverability) can be set at the top of the section “Reachability problem parameters” with the radio buttons. Each parameter (initial marking, target marking or predicates, conditions on transitions², optimized function) can be edited on a separate tab.

There are several options to enter the initial and target marking:

- They can be entered in the text boxes as a vector of integers separated with spaces.
- Using the “Options” button they can be edited in a separate window (Figure 5.6) where the ID, name and token count of each place is displayed.
- The actual token distribution of the net can be loaded from or into the *PetriDotNet* framework with the “Load from PDN” and “Show in PDN” buttons.

If the type of the problem is submarking coverability, the tab for predicates is visible (Figure 5.7) instead of the tab for the target marking. A new predicate can be added with the “Add predicate” button. The dialog seen in Figure 5.8 helps entering a predicate. The coefficient of each place can be set by selecting the name of the place in the combo box and entering the coefficient in the text box below. The type of the predicate (“ \geq ”, “ $=$ ”, “ \leq ”) can be set with the combo box in the top right corner. The right-hand side value can be

²The conditions on transitions are added directly to the ILP problem as a row.

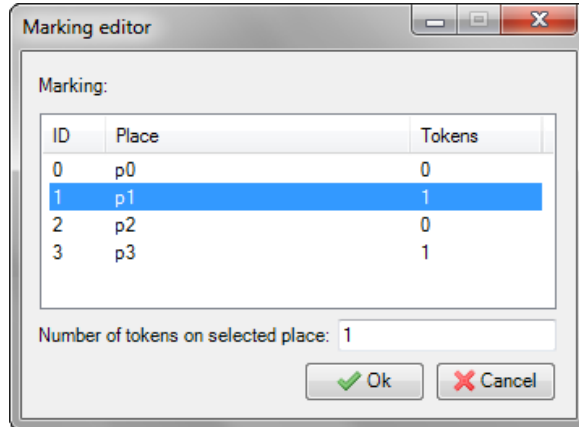


Figure 5.6: Marking editor dialog

entered in the text box below the type. The selected predicates can be removed with the “Remove” button. The list can be cleared by the “Remove all” button under “Options”. The option “Load tokens from PDN” under “Options” does the following: for each place p_i of the Petri net with $m(p_i) > 0$ a predicate of the form $m'(p_i) = m(p_i)$ is created, i.e., places with no tokens are ignored.

The conditions on transitions can be added similarly to predicates. The only difference is that these conditions correspond to transitions instead of places.

The optimized function of the ILP solver can be edited as a vector or with a helper dialog similar to the marking editor (Figure 5.6).

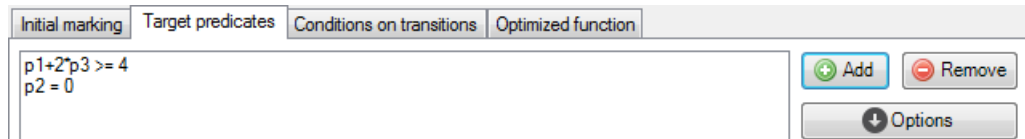


Figure 5.7: Predicates tab

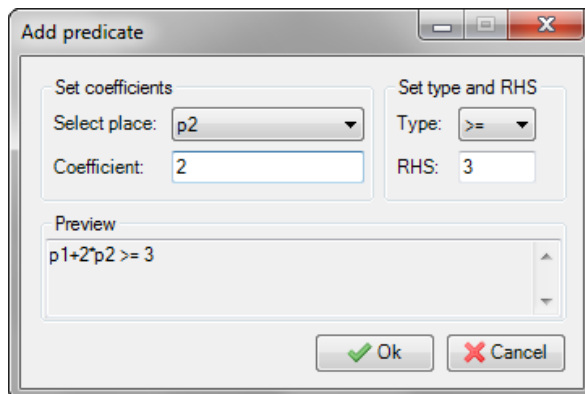


Figure 5.8: Predicate editor

All parameters can be written into an XML file with the “Save” button and loaded from an XML file with the “Load from file” button.

5.3.5 Configuration of the algorithm

The optimizations and the logging level of the algorithm can be configured by clicking the “Configuration” button. The following options are available (Figure 5.9):

- Level of logging: Sets the detailedness of logging. At level 0, only the solution is displayed, while at level 4, each detail is logged into the “Output” tab of the “Result” section.
- Generate state space: Sets whether the state space should be generated in the file “statespace.dot” in *GraphViz* format [17].
- Use stubborn sets: Enables the stubborn set optimization, which reduces the number of partial solutions by investigating dependencies and conflicts between transitions (Section 3.2.5).
- Store partial solutions: Enables the partial solution storing optimization, avoiding a partial solution to be processed multiple times (Section 3.2.5).
- Use subtree omission: Enables the subtree omission optimization, which reduces the number of partial solutions by ignoring the different order of transitions (Section 3.2.5).
- Filter partial solutions by T-invariants: Enables the T-invariant filtering optimization, which can avoid non-termination by detecting infinite loops in the abstraction refinement (Section 3.2.5).
- Try to involve distant T-invariants: Enables my new extension that tries to involve distant invariants when a partial solution is skipped by the filtering optimization (Section 4.3.2).
- Check state equation before finding increment constraints: Enables the new filtering criterion developed during our previous work (Section 4.3.1).
- Filter dead transitions: Enables filtering transitions that can never fire at the beginning of the algorithm³.

The reachability analysis can be started with the “Reachable?” button. It runs on a background thread and it can also be interrupted with the “Stop” button.

5.3.6 Examination of the result of the algorithm

The algorithm prints information depending on the level of logging in the “Output” tab of the result section. When the reachability analysis finished, detailed information can be seen in the “Result” tab. If a realizable solution is found, the plug-in displays the following items (Figure 5.10):

³Developed by Pál András Papp

- the initial marking,
- the marking reached by the solution,
- the solution vector,
- and the firing sequence realizing the solution.

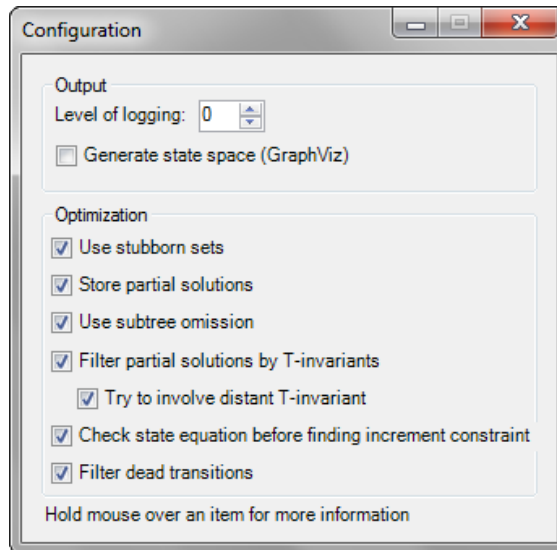


Figure 5.9: *Configuration dialog*

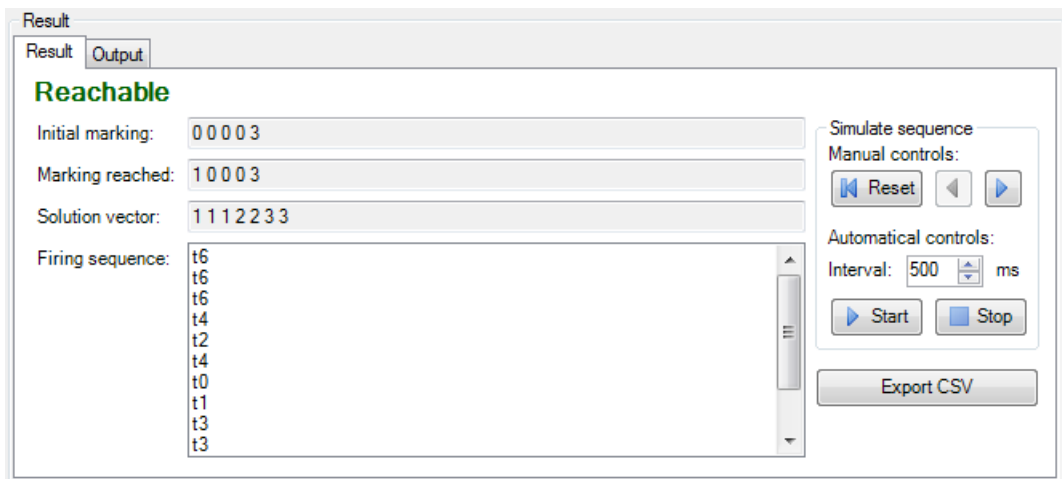


Figure 5.10: *Result of a reachable problem*

The firing sequence can be simulated automatically or manually with the playback controls in the “Simulate sequence” group. The result can also be exported into a CSV file with the “Export CSV” button. Clicking the button, a place selector dialog (Figure 5.11) appears. Each step of the firing sequence is written in the CSV file with the actual marking of the selected places.

If no full solution was found, the following cases are possible:

1. If some solutions were skipped by the T-invariant filtering optimization, the result is “Not decidable”.

2. If over-estimation occurred, the result is also “Not decidable”.
3. Otherwise, the result is “Not reachable”.

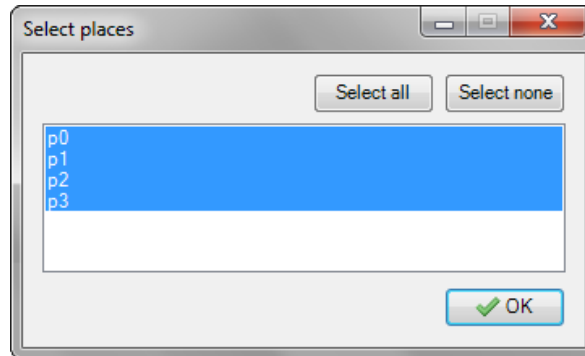


Figure 5.11: *Place selector dialog*

5.4 Development

In this section I introduce the ILP solver tool that I used (Section 5.4.1) and I present the details of the implementation (Section 5.4.2).

5.4.1 `lpsolve`

I solve the ILP problems using the open source *lpsolve* tool⁴ [16]. It has a C# interface, which can be downloaded from its website [16]. I only present the important parts in my work, a detailed documentation is available at the website [16].

The functionalities are provided in the static class `lpsolve` of the `lpsolve55` namespace. At first *lpsolve* has to be initialized with the method `init`, setting the path of the file *lpsolve55.dll*, which contains the implementation. A new LP problem can be created with the method `make_lp`. An LP problem is described by linear equalities or inequalities (constraints) over the variables. The first parameter of `make_lp` is the number of constraints (initially zero), while the second parameter is the number of variables (transitions in my case). The method returns the ID of the created problem (as an integer).

Constraints can be added to an LP problem with the method `add_constraint`, which has the following parameters:

- `lp`: ID of the LP problem, for which the constraint has to be added.
- `row`: Coefficients of the constraint in an array.
- `constr_type`: Type of the constraint. Possible values: equal (`EQ`), less-or-equal (`LE`), greater-or-equal (`GE`).
- `rh`: Right-hand side value of the constraint.

⁴*lpsolve* Version 5.5

The next step is to set the optimized function with the method `set_obj_fun`, taking the coefficients in an array. The algorithm wants to minimize the optimized function, which can be set by the method `set_minim`. Also, variables must be integer, which can be achieved by calling the method `set_int` on each variable.

The problem can be solved using the method `solve`, which returns whether it could solve the problem. If the method was successful, the result can be obtained by the function `get_variables`. After solving the problem, it must be deleted using the method `delete_lp`.

5.4.2 Classes

The plug-in (including the algorithm) is implemented in C# language. When creating the classes related to the algorithm, the main objective was not to strictly observe the basic principals of object oriented programming, but to let them easily represent a mathematic concept. The classes are categorized by their functionality and responsibility.

Main classes

- **CEGARplugin**: Connects the plug-in to the *PetriDotNet* framework by implementing the required interface of plug-ins.
- **Reachability**: Main class that solves the reachability problem. It gets the parameters of the problem and a reference of the caller in its constructor. The reachability analysis can be started with the method `IsReachable`. When the analysis is completed, the callback function of the caller is called.
- **ICaller**: Required interface for calling the algorithm. It has a callback function, which is called when the algorithm finishes the analysis.
- **ReachabilityFacade**: Facade class for using the plug-in without the graphical user interface.
- **Config**: Static class that stores the settings of the optimizations and the output.

Data structures

- **PetriNetMatrix**: Represents a Petri net with the following matrices: edge weights, incidence matrix, inhibitor arcs. In the inner representation, places and transitions are identified with a unique index starting from zero. This class provides several functions to retrieve information about the structure of the net. It can determine the set of enabled transitions under a given marking, and it can also calculate the marking when a transition or a sequence of transitions is fired. The stubborn set optimization is also implemented in this class: beneath the set of enabled transitions, the stubborn set of a marking can also be determined.
- **Vector**: Wrapper class for an array. Represents a mathematical vector. It has several helper methods, e.g., printing the vector and comparing it to another one.

- **MarkingRemainderPair**: A pair of two vectors: a marking and a remainder. Used by the subtree omission to detect markings reached by firing sequences only different in the order of transitions.
- **Predicate**: A class representing a row of a predicate of the form $Am \geq b$. It stores the coefficients of the row, the operator ($\geq, =, \leq$) and the right-hand side value.
- **ReachabilityParams**: Class for storing the parameters of the reachability problem: type (reachability or submarking coverability), initial marking, target marking or predicates, conditions on transitions, coefficients of the optimized function.
- **ReachabilityResult**: Class for storing the result of the reachability analysis: “reachable”, “not reachable” or “not decidable”. If the answer is “reachable”, the firing sequence is also provided.
- **SolSpaceLevel**: Represents a level of the solution space. Stores solutions of the state equation and partial solutions in a stack for backtrack purposes.
- **StEqSolution**: Represents a solution of the state equation. It also stores the constraints and the partial solutions generated from this solution.

Classes handling partial solutions

- **PartialSolution**: Represents a partial solution. It has a reference of the solution vector (with the constraints) and stores the firing sequence and the remainder vector. It also implements the T-invariant filtering optimization by the method `CanBeFilteredByTinvariants`.
- **MarkingTreeNode**: Represents a node in the tree for generating partial solutions.
- **PartialSolutionCatalog**: Implements the partial solution storing optimization.
- **TinvChain**: Represents a list of T-invariants, which is used by the new extension that involves distant invariants.

Classes handling constraints

- **IConstraint**: Common interface for constraints. Provides the array of coefficients, the operator and the right-hand side value for *lpsolve*.
- **IncrementConstraint**: Represents an increment constraint of the form $\sum n_i \cdot |t_i| \geq n$, implementing the interface **IConstraint**. Stores the coefficients and the right-hand side value. The operator is always “ \geq ”.
- **JumpConstraint**: Represents a jump constraint of the form $|t_i| < n$, implementing the interface **IConstraint**. Stores the index of the transition t_i and the right-hand side value. The operator is always “ \leq ”.

- **PredicateConstraint**: Represents the transformed version of a **Predicate**. Stores the coefficients of the transitions, the operator and the right-hand side value.
- **ConstrList**: Represents a list of constraints. It can transform jumps into increments and can also filter redundant constraints⁵.
- **IncrementBuilder**: Implements the increment constraint generating heuristic (Section 3.2.3). It uses the class **Graph** to build the dependency graph.

Dependency graph

- **Graph**: Represents the dependency graph used for generating increment constraints. It can store places and transitions as nodes and can find the source SCCs.
- **GraphNode**: Represents a node of the dependency graph. Stores the index of the place or transition and its neighbors.
- **STXtuple**: Represents a tuple of the form (P_i, T_i, X_i) .

Helper classes for `lpsolve`

- **lpsolve55**: Provides the interface of *lpsolve* and delegates the calls to the native implementation.
- **LpSolveException**: Custom class representing a possible exception by *lpsolve*.
- **LpSolveTool**: Helper class for using *lpsolve*. The function `Solve` builds the ILP problem using the incidence matrix and target marking (or predicates) and returns the solution if the problem is feasible. The function `StateEquationTest` implements the new filtering criterion discussed in Section 4.3.1.

Graphical User Interface

- **PluginForm**: Main window of the plug-in. It is divided into three sections: information about the net, parameters of the reachability problem and the result (Figure 5.4).
- **AboutCegarTool**: Displays information about the authors and version of the plug-in.
- **AddCondTrans**: Dialog for adding conditions on transitions.
- **AddPredicate**: Dialog for adding predicates (Figure 5.8).
- **ConfigForm**: Dialog for configuring the plug-in (Figure 5.9).
- **ConsoleHandler**: Helper class for logging information in the main window during the execution of the algorithm.

⁵E.g., the constraint $|t_1| < 3$ is redundant when a constraint $|t_1| < 2$ is also present.

- **DisplayNames**: Displays places and transitions with their IDs and names (Figure 5.5). The algorithm uses IDs in the inner representation, but the user can give names to the places or transitions.
- **GraphViz**: Helper class for creating solution space graphs for *GraphViz* [17].
- **MarkingEditor**: Dialog for editing the initial and target marking (Figure 5.6).
- **PlaceSelector**: Dialog for selecting places that should be included in the exported statistics (Figure 5.11).
- **IOHelper**: Helper class for I/O operations such as reading or writing the parameters of the reachability problem from or into files and the user interface.

Chapter 6

Evaluation

In this chapter I present my measurement results for well-known models. Section 6.1 shows how the runtime of the algorithm scales for several models with a given parameter. In Section 6.2 I compare my algorithm to a different type of verification algorithm, namely the saturation method.

The tests were taken with the following configuration:

- Hardware: Intel Core i5 M430 2,27Ghz processor and 3 GB RAM
- Operating system: Windows 7 x32
- .NET Framework version 4.0
- *PetriDotNet* framework version 1.3.4804

6.1 Scalability

In this section I present measurement results for well-known models. The measurements focused on the scalability of the runtime with a given parameter of the model. In some cases the parameter affects only the token distribution of the net, while in other cases it also affects the structure of the model.

6.1.1 Counter

The Counter model represents a simple n bit binary counter. The model contains inhibitor arcs. The problem solved by the algorithm is to count from 0 to $2^n - 1$, where n is the parameter. The results can be seen in Table 6.1 and Figure 6.1. The runtime clearly scales exponentially with the parameter, which is not surprising, since the length of the firing sequence solving the problem is also exponential of n .

6.1.2 Dining philosophers

The Dining philosophers model [18] is often used to illustrate the problems of parallel programming and mutual exclusion. There are n philosophers around a circular table. Each philosopher has a plate and there is a fork between each two plates. A philosopher

can eat if he has a fork in both hands. Since two neighbors share a fork, at most $n/2$ philosophers can eat at the same time. Each philosopher is either thinking or eating. If a philosopher gets hungry, he grabs the forks next to him and eats. After eating, he puts back the forks. There is a possibility for deadlock if all the philosophers get hungry at the same time and they all grab one fork. In this case none of them can eat, therefore they will not put back the forks.

The problem solved by the algorithm is to reach a state, where every second philosopher is eating. The results can be seen in Table 6.2 and Figure 6.2. This model has a very large structure (the incidence matrix has $24n^2$ elements) therefore, finding the solution vector with the ILP solver is already a hard problem.

Parameter	Runtime
4	0,011 s
8	0,02 s
12	0,038 s
14	0,096 s
16	0,195 s
18	0,994 s
20	3,726 s

Table 6.1: Measurement results for the Counter model

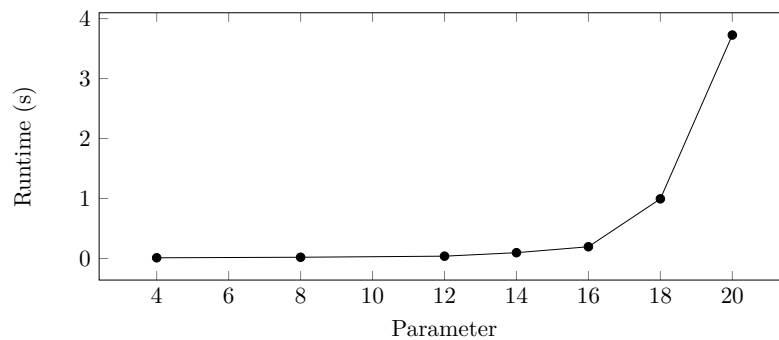


Figure 6.1: Measurement results for the Counter model

Parameter	Runtime
10	0,092 s
20	0,241 s
30	0,488 s
50	1,468 s
100	9,754 s
200	83,357 s

Table 6.2: Measurement results for the Dining philosophers model

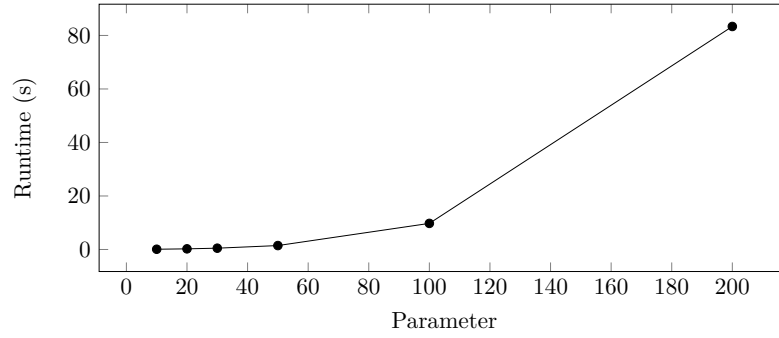


Figure 6.2: *Measurement results for the Dining philosophers model*

6.1.3 FMS

The FMS model [19] represents a flexible manufacturing system where different types of parts are assembled together. The parameter of this model is the number of parts to be assembled (n), which determines the initial marking. The structure of the net is the same for all n . The results can be seen in Table 6.3 and Figure 6.3. Since the structure of the net does not change, the size of the abstract model is constant. However, the length of the firing sequence solving the problem grows linearly, which yields a linear scalability of the runtime.

Parameter	Runtime
10	0,046 s
50	0,053 s
100	0,058 s
200	0,078 s
400	0,115 s
800	0,191 s
1600	0,321 s
3200	0,638 s
6400	1,28 s
12800	2,571 s

Table 6.3: *Measurement results for the FMS model*

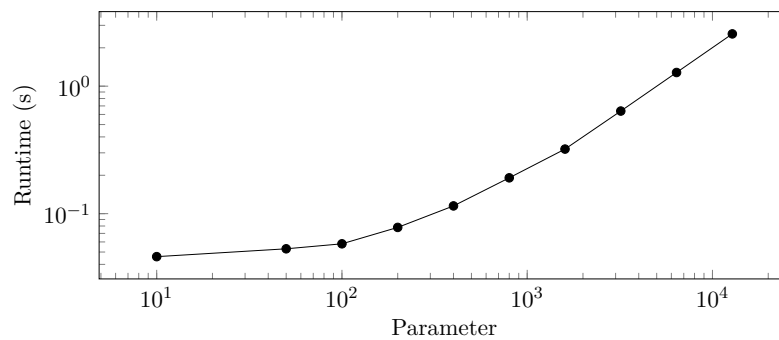


Figure 6.3: *Measurement results for the FMS model*

6.1.4 Kanban

The Kanban model represents a production scheduling method. The parameter of this model determines the initial marking, the structure is the same for all n . The results can be seen in Table 6.4 and Figure 6.4. Although the size of the model does not change, the runtime scales exponentially with the parameter. I experienced that the algorithm can find a realizable solution vector quickly, but it examines many partial solutions before it finds a full solution, i.e., there are many dead-ends in the partial solution tree.

Parameter	Runtime
10	0,352 s
13	1,147 s
16	3,424 s
19	8,040 s
22	17,51 s
25	35,061 s
28	64,947 s

Table 6.4: Measurement results for the Kanban model

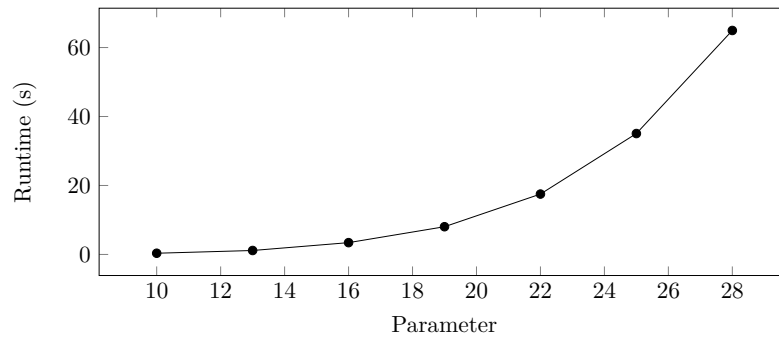


Figure 6.4: Measurement results for the Kanban model

6.1.5 Slotted ring

The Slotted ring model represents a network protocol. The parameter is the number of participants in the network. The results can be seen in Table 6.5 and Figure 6.5. Although the size of the model grows, the ILP solver can handle this model well and this yields a polynomial runtime.

Parameter	Runtime
10	0,207 s
20	0,571 s
30	1,206 s
35	1,585 s
50	3.461 s

Table 6.5: Measurement results for the Slotted ring model

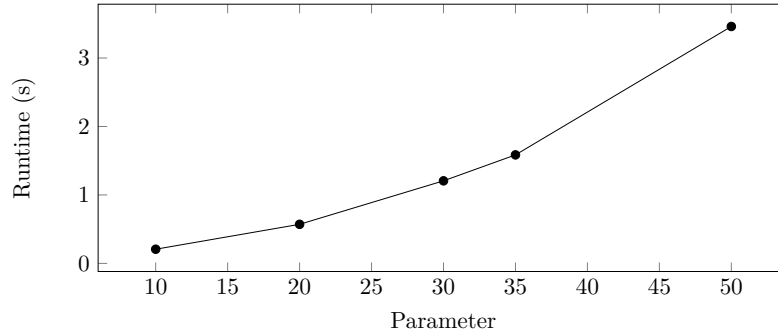


Figure 6.5: Measurement results for the Slotted ring model

6.2 Comparison to the saturation algorithm

In this section I compare my implementation of the CEGAR algorithm to the saturation method, which is also implemented in the *PetriDotNet* framework [20]. The results can be seen in Table 6.6. “TO” refers to an unacceptable runtime ($> 600s$).

Model and parameter	CEGAR	Saturation
Philosophers-10	0,092	0,01 s
Philosophers-20	0,241	0,02 s
Philosophers-30	0,488	0,03 s
Philosophers-50	1,468	0,03 s
FMS-10	0,046 s	0,06 s
FMS-50	0,053 s	1,09 s
FMS-100	0,058 s	8,03 s
FMS-200	0,078 s	69,72 s
FMS-400	0,115 s	TO
FMS-800	0,191 s	TO
Kanban-10	0,352 s	0 s
Kanban-16	3,424 s	0,01 s
Kanban-19	8,040 s	0,03 s
Kanban-25	35,061 s	0,05 s
SlottedRing-10	0,207 s	0,06 s
SlottedRing-20	0,571 s	0,042 s
SlottedRing-30	1,206 s	1,47 s
SlottedRing-35	1,585 s	2,43 s
SlottedRing-50	3.461 s	6,82 s

Table 6.6: Comparison of the CEGAR and saturation algorithm

The two algorithms are complementers: saturation performs better for models with large structures, since solving the ILP problem has a big cost. However, for models with simple structure and large state space, the CEGAR approach is more effective. Moreover, CEGAR can also handle infinite problems.

Chapter 7

Conclusions

The *counterexample guided abstraction refinement* (CEGAR) is a new and promising formal analysis technique. In this thesis I developed and evaluated improvements to a recently published CEGAR algorithm for the reachability analysis of Petri nets. These improvements have both theoretical and practical importance. On the theoretical side, I have continued my previous work, where I (together with my colleagues) proved the incorrectness and incompleteness of the algorithm and suggested improvements to overcome some of the problems. On the practical side, the developed extensions significantly widen the applicability of the CEGAR approach, as well as improve its performance.

In my thesis work I studied the correctness and completeness of the CEGAR algorithm further, and proved that one of the optimizations still creates the possibility of incorrectness. I suggested a structural method for detecting such situations, which is independent from the optimizations. Furthermore, I developed a new method that can not only detect incorrectness, but can also help to find a solution in such cases. I also improved the iteration strategy of the algorithm in order to extend the set of decidable problems.

I implemented the algorithm with the new contributions in the *PetriDotNet* framework and measured its performance on well-known models. I compared my implementation of the CEGAR algorithm to the so-called saturation method, which is also a new and very efficient analysis technique. The results have shown that the two algorithms can complement each other: saturation performs better for models with large structures, however, for models with simple structure and large state space, the CEGAR approach is more effective. Another important advantage of CEGAR over saturation is that it can handle even infinite problems.

During my preparation of this thesis I successfully completed all of the specified objectives. I presented the CEGAR algorithm for Petri nets (see Section 3.2). I examined the shortcomings of the algorithm, and suggested improvements (see Sections 4.2 and 4.3). I implemented the algorithm and evaluated its performance by measurements (see Chapters 5 and 6).

Even though all the objectives were met, there are still several opportunities for further research. In some special situations, the algorithm can detect incorrectness, but cannot decide the problem. There are also problems that even the extended iteration strategy

cannot solve. Thus, an important result would be if either the algorithm could be extended to handle these situations, or the impossibility of making the algorithm complete could be formally proven. Another promising research direction is indicated by the measurement results. They show that the algorithm performs poorly for certain models, therefore new optimizations are required to improve the performance of the CEGAR approach.

Acknowledgment

I would like to express my acknowledgement to my family, my supervisors, my girlfriend and everybody else who supported me during this work.

List of Figures

2.1	Example net modeling a chemical process	11
2.2	Example Petri net	11
2.3	Example for an unrealizable and a realizable solution	12
2.4	Example for T-invariants	13
2.5	Example net with inhibitor arcs	14
2.6	Submarking coverability example	15
3.1	CEGAR flowchart	18
3.2	Petri net specific CEGAR flowchart	19
3.3	Example nets for jump and increment constraints	21
3.4	Solution space of the state equation	21
3.5	Partial solution tree example	22
3.6	If $at\sigma u$ and $au\sigma t$ can both fire, then only one of the subtrees after \hat{m} needs to be processed.	27
3.7	A complex example showing several aspects of the algorithm	28
3.8	Dependency graphs of PS_0 and PS_1	29
3.9	Solution space of the example seen in Figure 3.7	30
4.1	Proof of the incorrectness of the algorithm	33
4.2	Dependency graph of the partial solution PS	33
4.3	Part of the solution space showing how solutions can be found in case of over-estimation	34
4.4	In some special cases, even estimating $n = 1$ does not help finding a solution.	35
4.5	A net, where over-estimation cannot be detected if subtree omission is used	36
4.6	A counterexample of completeness	38
4.7	T-invariant example	40
4.8	A complex example for involving distant invariants	41
4.9	Solution space of the example on distant invariants	42
4.10	Limitations of the new approach on distant invariants	43
5.1	Main screen of PetriDotNet	49
5.2	Overview of the architecture	49
5.3	Files and directories of the plug-in	50
5.4	Main window of the CEGAR plug-in	50

5.5	Place names with IDs	51
5.6	Marking editor dialog	52
5.7	Predicates tab	52
5.8	Predicate editor	52
5.9	Configuration dialog	54
5.10	Result of a reachable problem	54
5.11	Place selector dialog	55
6.1	Measurement results for the Counter model	61
6.2	Measurement results for the Dining philosophers model	62
6.3	Measurement results for the FMS model	62
6.4	Measurement results for the Kanban model	63
6.5	Measurement results for the Slotted ring model	64

List of Tables

6.1	Measurement results for the Counter model	61
6.2	Measurement results for the Dining philosophers model	61
6.3	Measurement results for the FMS model	62
6.4	Measurement results for the Kanban model	63
6.5	Measurement results for the Slotted ring model	63
6.6	Comparison of the CEGAR and saturation algorithm	64

Bibliography

- [1] H. Wimmel and K. Wolf, “Applying CEGAR to the Petri Net State Equation,” in *Tools and Algorithms for the Construction and Analysis of Systems, 17th International Conference, TACAS 2010 Proceedings* (P. A. Abdulla and K. R. M. Leino, eds.), vol. 6605 of *Lecture Notes in Computer Science*, pp. 224–238, Springer, 2011.
- [2] A. Hajdu and Z. Mártonka, “Diszkrét dinamikus rendszerek viselkedésének felderítése ellenpélda alapú absztrakció finomítás (CEGAR) segítségével,” 2012.
- [3] A. Hajdu, A. Vörös, T. Bartha, and Z. Mártonka, “Extensions to the CEGAR Approach on Petri Nets,” in *Proceedings of the 13th Symposium on Programming Languages and Software Tools, SPLST’2013,*, pp. 274–288, 2013.
- [4] T. Murata, “Petri Nets: Properties, analysis and applications,” *Proceedings of the IEEE*, vol. 77, pp. 541–580, April 1989.
- [5] P. Chrzastowski-Wachtel, “Testing Undecidability of the Reachability in Petri Nets with the Help of 10th Hilbert Problem,” in *Application and Theory of Petri Nets 1999* (S. Donatelli and J. Kleijn, eds.), vol. 1639 of *Lecture Notes in Computer Science*, pp. 690–690, Springer, 1999.
- [6] Pataricza András, ed., *Formális módszerek az informatikában*. Typotex, 2. kiadás, 2005.
- [7] N. Busi, “Analysis issues in Petri Nets with inhibitor arcs,” *Theoretical Computer Science*, vol. 275, pp. 127–177, Mar. 2002.
- [8] J. Esparza and S. Melzer, “Verification of safety properties using integer programming: Beyond the state equation,” *Formal Methods in System Design*, vol. 16, no. 2, pp. 159–189, 2000.
- [9] E. W. Mayr, “An algorithm for the general Petri Net reachability problem,” in *Proceedings of the thirteenth annual ACM symposium on Theory of computing, STOC ’81*, (New York, NY, USA), pp. 238–246, ACM, 1981.
- [10] R. Lipton, *The Reachability Problem Requires Exponential Space*. Research report, Department of Computer Science, Yale University, 1976.

- [11] G. Ciardo, G. Lüttgen, and R. Siminiceanu, “Saturation: An efficient iteration strategy for symbolic state-space generation,” in *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001*, (London, UK, UK), pp. 328–342, Springer-Verlag, 2001.
- [12] G. B. Dantzig and M. N. Thapa, *Linear programming 1: introduction*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1997.
- [13] A. Valmari and H. Hansen, “Can stubborn sets be optimal?,” in *Applications and Theory of Petri Nets* (J. Lilius and W. Penczek, eds.), vol. 6128 of *Lecture Notes in Computer Science*, pp. 43–62, Springer, 2010.
- [14] L. M. Kristensen, K. Schmidt, and A. Valmari, “Question-guided stubborn set methods for state properties,” *Form. Methods Syst. Des.*, vol. 29, pp. 215–251, Nov. 2006.
- [15] “Homepage of the *PetriDotNet* framework.
<http://petridotnet.inf.mit.bme.hu/>.” [Online; accessed 07-12-2013].
- [16] “Homepage of the *lpsolve* tool.
<http://sourceforge.net/projects/lpsolve/>.” [Online; accessed 07-12-2013].
- [17] “Homepage of *GraphViz*.
<http://www.graphviz.org/>.” [Online; accessed 07-12-2013].
- [18] E. Dijkstra, “Hierarchical ordering of sequential processes,” *Acta Informatica*, vol. 1, no. 2, pp. 115–138, 1971.
- [19] G. Ciardo, Y. Zhao, and X. Jin, “Transactions on Petri Nets and Other Models of Concurrency V,” ch. Ten Years of Saturation: A Petri Net Perspective, pp. 51–95, Berlin, Heidelberg: Springer-Verlag, 2012.
- [20] A. Vörös, D. Darvas, and T. Bartha, “Bounded Saturation Based CTL Model Checking,” in *Proceedings of the 12th Symposium on Programming Languages and Software Tools, SPLST’11*, 2011.